
Charm++ Documentation

PPL

Oct 25, 2021

1	Charm++ Quickstart	3
2	The Charm++ Parallel Programming System	7
3	Adaptive MPI (AMPI)	185
4	Fortran90 Bindings for Charm++	215
5	Converse and Charm++ Libraries	221
6	Frequently Asked Questions	233
7	Threaded Charm++ (TCharm)	263
8	Finite Element Method (FEM) Framework	273
9	Iterative Finite Element Matrix (IFEM) Framework	313
10	NetFEM Framework	321
11	Multiblock Framework	327
12	Parallel Object-Oriented Simulation Environment (POSE)	341
13	Converse Programming	353
14	Converse Extensions Library	387
15	Projections	417
16	Charm++ Debugger	445
17	Charisma	459
18	Parallel Framework for Unstructured Meshes (ParFUM)	471
19	Charm++/Converse license	483
	Bibliography	487

- Charm++ Tutorial: <https://charmplusplus.org/tutorial/>
- Charm++ Development: <https://github.com/UIUC-PPL/charm>
- Nightly regression tests: <https://charm.cs.illinois.edu/autobuild/cur/>
- Charm++ discussion: <https://github.com/UIUC-PPL/charm/discussions>

This section gives a concise overview of running your first Charm++ application.

1.1 Installing Charm++

To download the latest Charm++ release, run:

```
$ wget https://charm.cs.illinois.edu/distrib/charm-latest.tar.gz
$ tar xzf charm-latest.tar.gz
```

To download the development version of Charm++, run:

```
$ git clone https://github.com/UIUC-PPL/charm
```

To build Charm++, use the following commands:

```
$ cd charm
$ ./build charm++ netlrts-linux-x86_64 --with-production -j4
```

This is the recommended version to install Charm++ on Linux systems. For macOS, substitute “linux” with “darwin”, and on ARM64 machines, replace “x86_64” with “arm8”. For advanced compilation options, please see Section 2.6.1 of the manual.

1.2 Parallel “Hello World” with Charm++

The basic unit of computation in Charm++ is a **chare**, which is a C++ object. Chares have **entry methods** that can be invoked asynchronously. A Charm++ application consists of collections of chares (such as chare arrays) distributed among the processors of the system.

Each chare has a **proxy** associated to it, through which other chares can invoke entry methods. This proxy is exposed through the **thisProxy** member variable, which can be sent to other chares, allowing them to invoke entry methods on this chare.

Each Charm++ application consists of at least two files, a *Charm interface* (.ci) file, and a normal C++ file. The interface file describes the parallel interface of the application (such as chares, chare arrays, and entry methods), while the C++ files implement its behavior. Please see Section 2.2.1 of the manual for more information about the program structure.

In this section, we present a parallel *Hello World* example, consisting of the files `hello.ci` and `hello.cpp`.

1.2.1 The hello.ci File

The `hello.ci` file contains a `mainchare`, which starts and ends execution, and a `Hello` chare array, whose elements print the “Hello World” message. Compiling this file creates C++ header files (`hello.decl.h` and `hello.def.h`) that can be included in your C++ files.

```
mainmodule hello {
  mainchare Main {
    // Main's entry methods
    entry Main(CkArgMsg *m);
    entry void done();
  };
  array [1D] Hello {
    // Hello's entry methods
    entry Hello();
    entry void SayHi();
  };
};
```

1.2.2 The hello.cpp File

The `hello.cpp` file contains the implementation of the `mainchare` and chare array declared in the `hello.ci` file above.

```
#include "hello.decl.h" // created from hello.ci file above

/*readonly*/ CProxy_Main mainProxy;
constexpr int nElem = 8;

/*mainchare*/
class Main : public CBase_Main
{
public:
  Main(CkArgMsg* m)
  {
    //Start computation
    CkPrintf("Running Hello on %d processors with %d elements.\n", CkNumPes(), nElem);
    CProxy_Hello arr = CProxy_Hello::ckNew(nElem); // Create a new chare array with
    ↪nElem elements
    mainProxy = thisProxy;
    arr[0].SayHi(0);
  };

  void done()
  {
    // Finish computation
    CkPrintf("All done.\n");
    CkExit();
  }
};
```

(continues on next page)

(continued from previous page)

```

    };
};

/*array [1D]*/
class Hello : public CBase_Hello
{
public:
    Hello() {}

    void SayHi()
    {
        // thisIndex stores the element's array index
        CkPrintf("PE %d says: Hello world from element %d.\n", CkMyPe(), thisIndex);
        if (thisIndex < nElem - 1) {
            thisProxy[thisIndex + 1].SayHi(); // Pass the hello on
        } else {
            mainProxy.done(); // We've been around once -- we're done.
        }
    }
};

#include "hello.def.h" // created from hello.ci file above

```

1.2.3 Compiling the Example

Charm++ has a compiler wrapper, `charmcc`, to compile Charm++ applications. Please see Section 2.6.2 for more information about `charmcc`.

```

$ charm/bin/charmcc hello.ci # creates hello.def.h and hello.decl.h
$ charm/bin/charmcc hello.cpp -o hello

```

1.2.4 Running the Example

Charm++ applications are started via `charmrun`, which is automatically created by the `charmcc` command above. Please see Section 2.6.3 for more information about `charmrun`.

To run the application on two processors, use the following command:

```

$ ./charmrun +p2 ./hello
Charmrun> scalable start enabled.
Charmrun> started all node programs in 1.996 seconds.
Charm++> Running in non-SMP mode: 1 processes (PEs)
Converse/Charm++ Commit ID: v6.9.0-172-gd31997cce
Charm++> scheduler running in netpoll mode.
CharmLB> Load balancer assumes all CPUs are same.
Charm++> Running on 1 hosts (1 sockets x 4 cores x 2 PUs = 8-way SMP)
Charm++> cpu topology info is gathered in 0.000 seconds.
Running Hello on 2 processors with 8 elements.
PE 0 says: Hello world from element 0.
PE 0 says: Hello world from element 1.
PE 0 says: Hello world from element 2.
PE 0 says: Hello world from element 3.
PE 1 says: Hello world from element 4.

```

(continues on next page)

(continued from previous page)

```
PE 1 says: Hello world from element 5.  
PE 1 says: Hello world from element 6.  
PE 1 says: Hello world from element 7.  
All done  
[Partition 0][Node 0] End of program
```

1.3 Where to go From Here

- The `tests/charm++/simplearrayhello` folder in the Charm++ distribution has a more comprehensive example, from which the example in this file was derived.
- The main Charm++ manual (<https://charm.readthedocs.io/>) contains more information about developing and running Charm++ applications.
- Charm++ has lots of other features, such as chare migration, load balancing, and checkpoint/restart. The main manual has more information about them.
- AMPI (<https://charm.readthedocs.io/en/latest/mpi/manual.html>) is an implementation of MPI on top of Charm++, allowing MPI applications to run on the Charm++ runtime mostly unmodified.
- Charm4py (<https://charm4py.readthedocs.io>) is a Python package that enables development of Charm++ applications in Python.

The Charm++ Parallel Programming System

Contents

- *The Charm++ Parallel Programming System*
 - *Basic Concepts*
 - * *Programming Model*
 - * *Execution Model*
 - * *Proxies and the charm interface file*
 - * *Machine Model*
 - *Basic Charm++ Programming*
 - * *Program Structure, Compilation and Utilities*
 - * *Basic Syntax*
 - * *Chare Arrays*
 - * *Structured Control Flow: Structured Dagger*
 - * *Serialization Using the PUP Framework*
 - * *Load Balancing*
 - * *Processor-Aware Chare Collections*
 - * *Initializations at Program Startup*
 - *Advanced Programming Techniques*
 - * *Optimizing Entry Method Invocation*
 - * *Callbacks*
 - * *Waiting for Completion*

- * *More Chare Array Features*
- * *Sections: Subsets of a Chare Array/Group*
- * *Chare and Message Inheritance*
- * *Generic and Meta Programming with Templates*
- * *Collectives*
- * *Serializing Complex Types*
- * *Querying Network Topology*
- * *Physical Node API*
- * *Checkpoint/Restart-Based Fault Tolerance*
- * *Support for Loop-level Parallelism*
- * *GPU Support*
- * *Charm-MPI Interoperation*
- * *Interoperation with Kokkos*
- * *Interoperation with RAJA*
- * *Partitioning in Charm++*
- *Expert-Level Functionality*
 - * *Tuning and Developing Load Balancers*
 - * *Dynamic Code Injection*
 - * *Intercepting Messages via Delegation*
- *Experimental Features*
 - * *Control Point Automatic Tuning*
 - * *Malleability: Shrink/Expand Number of Processors*
- *Appendix*
 - * *Installing Charm++*
 - * *Compiling Charm++ Programs*
 - * *Running Charm++ Programs*
 - * *Reserved words in .c.i files*
 - * *Performance Tracing for Analysis*
 - * *Debugging*
 - * *History*
 - * *Acknowledgements*

2.1 Basic Concepts

Charm++ is a C++-based parallel programming system, founded on the migratable-objects programming model, and supported by a novel and powerful adaptive runtime system. It supports both irregular as well as regular applications,

and can be used to specify task-parallelism as well as data parallelism in a single application. It automates dynamic load balancing for task-parallel as well as data-parallel applications, via separate suites of load-balancing strategies. Via its message-driven execution model, it supports automatic latency tolerance, modularity and parallel composition. Charm++ also supports automatic checkpoint/restart, as well as fault tolerance based on distributed checkpoints.

Charm++ is a production-quality parallel programming system used by multiple applications in science and engineering on supercomputers as well as smaller clusters around the world. Currently the parallel platforms supported by Charm++ are the IBM BlueGene/Q and OpenPOWER systems, Cray XE, XK, and XC systems, Omni-Path and Infiniband clusters, single workstations and networks of workstations (including x86 (running Linux, Windows, MacOS)), etc. The communication protocols and infrastructures supported by Charm++ are UDP, MPI, OFI, UCX, Infiniband, uGNI, and PAMI. Charm++ programs can run without changing the source on all these platforms. Charm++ programs can also interoperate with MPI programs (Section 2.3.15). Please see the Installation and Usage section for details about installing, compiling and running Charm++ programs (Section 2.6.1).

2.1.1 Programming Model

The key feature of the migratable-objects programming model is *over-decomposition*: The programmer decomposes the program into a large number of work units and data units, and specifies the computation in terms of creation of and interactions between these units, without any direct reference to the processor on which any unit resides. This empowers the runtime system to assign units to processors, and to change the assignment at runtime as necessary. Charm++ is the main (and early) exemplar of this programming model. AMPI is another example within the Charm++ family of the same model.

2.1.2 Execution Model

A basic unit of parallel computation in Charm++ programs is a *chare*. A chare is similar to a process, an actor, an ADA task, etc. At its most basic level, it is just a C++ object. A Charm++ computation consists of a large number of chares distributed on available processors of the system, and interacting with each other via asynchronous method invocations. Asynchronously invoking a method on a remote object can also be thought of as sending a “message” to it. So, these method invocations are sometimes referred to as messages. (besides, in the implementation, the method invocations are packaged as messages anyway). Chares can be created dynamically.

Conceptually, the system maintains a “work-pool” consisting of seeds for new chares, and messages for existing chares. The Charm++ runtime system (*Charm RTS*) may pick multiple items, non-deterministically, from this pool and execute them, with the proviso that two different methods cannot be simultaneously executing on the same chare object (say, on different processors). Although one can define a reasonable theoretical operational semantics of Charm++ in this fashion, a more practical description of execution is useful to understand Charm++. A Charm++ application’s execution is distributed among Processing Elements (PEs), which are OS threads or processes depending on the selected Charm++ build options. (See section 2.1.4 for a precise description.) On each PE, there is a scheduler operating with its own private pool of messages. Each instantiated chare has one PE which is where it currently resides. The pool on each PE includes messages meant for Chares residing on that PE, and seeds for new Chares that are tentatively meant to be instantiated on that PE. The scheduler picks a message, creates a new chare if the message is a seed (i.e. a constructor invocation) for a new Chare, and invokes the method specified by the message. When the method returns control back to the scheduler, it repeats the cycle. I.e. there is no pre-emptive scheduling of other invocations.

When a chare method executes, it may create method invocations for other chares. The Charm Runtime System (RTS) locates the PE where the targeted chare resides, and delivers the invocation to the scheduler on that PE.

Methods of a chare that can be remotely invoked are called *entry* methods. Entry methods may take serializable parameters, or a pointer to a message object. Since chares can be created on remote processors, obviously some constructor of a chare needs to be an entry method. Ordinary entry methods¹ are completely non-preemptive- Charm++ will not interrupt an executing method to start any other work, and all calls made are asynchronous.

¹ “Threaded” or “synchronous” methods are different. But even they do not lead to pre-emption; only to cooperative multi-threading

Charm++ provides dynamic seed-based load balancing. Thus location (processor number) need not be specified while creating a remote chare. The Charm RTS will then place the remote chare on a suitable processor. Thus one can imagine chare creation as generating only a seed for the new chare, which may *take root* on some specific processor at a later time.

Chares can be grouped into collections. The types of collections of chares supported in Charm++ are: *chare-arrays*, *chare-groups*, and *chare-nodegroups*, referred to as *arrays*, *groups*, and *nodegroups* throughout this manual for brevity. A Chare-array is a collection of an arbitrary number of migratable chares, indexed by some index type, and mapped to processors according to a user-defined map group. A group (nodegroup) is a collection of chares, with exactly one member element on each PE (“node”).

Charm++ does not allow global variables, except readonly variables (see 2.2.2). A chare can normally only access its own data directly. However, each chare is accessible by a globally valid name. So, one can think of Charm++ as supporting a *global object space*.

Every Charm++ program must have at least one mainchare. Each mainchare is created by the system on processor 0 when the Charm++ program starts up. Execution of a Charm++ program begins with the Charm RTS constructing all the designated mainchares. For a mainchare named X, execution starts at constructor X() or X(CkArgMsg *) which are equivalent. Typically, the mainchare constructor starts the computation by creating arrays, other chares, and groups. It can also be used to initialize shared readonly objects.

Charm++ program execution is terminated by the CkExit call. Like the exit system call, CkExit never returns, and it optionally accepts an integer value to specify the exit code that is returned to the calling shell. If no exit code is specified, a value of zero (indicating successful execution) is returned. The Charm RTS ensures that no more messages are processed and no entry methods are called after a CkExit. CkExit need not be called on all processors; it is enough to call it from just one processor at the end of the computation.

As described so far, the execution of individual Chares is “reactive”: When method A is invoked the chare executes this code, and so on. But very often, chares have specific life-cycles, and the sequence of entry methods they execute can be specified in a structured manner, while allowing for some localized non-determinism (e.g. a pair of methods may execute in any order, but when they both finish, the execution continues in a pre-determined manner, say executing a 3rd entry method). To simplify expression of such control structures, Charm++ provides two methods: the structured dagger notation (Section 2.2.4), which is the main notation we recommend you use. Alternatively, you may use threaded entry methods, in combination with *futures* and *sync* methods (See 2.3.3). The threaded methods run in light-weight user-level threads, and can block waiting for data in a variety of ways. Again, only the particular thread of a particular chare is blocked, while the PE continues executing other chares.

The normal entry methods, being asynchronous, are not allowed to return any value, and are declared with a void return type. However, the *sync* methods are an exception to this. They must be called from a threaded method, and so are allowed to return (certain types of) values.

2.1.3 Proxies and the charm interface file

To support asynchronous method invocation and global object space, the RTS needs to be able to serialize (“marshall”) the parameters, and be able to generate global “names” for chares. For this purpose, programmers have to declare the chare classes and the signature of their entry methods in a special “.ci” file, called an interface file. Other than the interface file, the rest of a Charm++ program consists of just normal C++ code. The system generates several classes based on the declarations in the interface file, including “Proxy” classes for each chare class. Those familiar with various component models (such as CORBA) in the distributed computing world will recognize “proxy” to be a dummy, standin entity that refers to an actual entity. For each chare type, a “proxy” class exists. The methods of this “proxy” class correspond to the remote methods of the actual class, and act as “forwarders”. That is, when one invokes a method on a proxy to a remote object, the proxy marshalls the parameters into a message, puts adequate information about the target chare on the envelope of the message, and forwards it to the remote object. Individual chares, chare array, groups, node-groups, as well as the individual elements of these collections have a such a proxy. Multiple methods for obtaining such proxies are described in the manual. Proxies for each type of entity in Charm++

have some differences among the features they support, but the basic syntax and semantics remain the same - that of invoking methods on the remote object by invoking methods on proxies.

The following sections provide detailed information about various features of the Charm++ programming system. Part I, “Basic Usage”, is sufficient for writing full-fledged applications. Note that only the last two chapters of this part involve the notion of physical processors (cores, nodes, ..), with the exception of simple query-type utilities (Section 2.2.1). We strongly suggest that all application developers, beginners and experts alike, try to stick to the basic language to the extent possible, and use features from the advanced sections only when you are convinced they are essential. (They are useful in specific situations; but a common mistake we see when we examine programs written by beginners is the inclusion of complex features that are not necessary for their purpose. Hence the caution). The advanced concepts in the Part II of the manual support optimizations, convenience features, and more complex or sophisticated features.²

2.1.4 Machine Model

At its basic level, Charm++ machine model is very simple: Think of each chare as a separate processor by itself. The methods of each chare can access its own instance variables (which are all private, at this level), and any global variables declared as *readonly*. It also has access to the names of all other chares (the “global object space”), but all that it can do with that is to send asynchronous remote method invocations towards other chare objects. (Of course, the instance variables can include as many other regular C++ objects that it “has”; but no chare objects. It can only have references to other chare objects).

In accordance with this vision, the first part of the manual (up to and including the chapter on load balancing) has almost no mention of entities with physical meanings (cores, nodes, etc.). The runtime system is responsible for the magic of keeping closely communicating objects on nearby physical locations, and optimizing communications within chares on the same node or core by exploiting the physically available shared memory. The programmer does not have to deal with this at all. The only exception to this pure model in the basic part are the functions used for finding out which “processor” an object is running on, and for finding how many total processors are there.

However, for implementing lower level libraries, and certain optimizations, programmers need to be aware of processors. In any case, it is useful to understand how the Charm++ implementation works under the hood. So, we describe the machine model, and some associated terminology here.

In terms of physical resources, we assume the parallel machine consists of one or more *nodes*, where a node is a largest unit over which cache coherent shared memory is feasible (and therefore, the maximal set of cores per which a single process *can* run. Each node may include one or more processor chips, with shared or private caches between them. Each chip may contain multiple cores, and each core may support multiple hardware threads (SMT for example).

Charm++ recognizes two logical entities: a PE (processing element) and a logical node, or simply “node”. In a Charm++ program, a PE is a unit of mapping and scheduling: each PE has a scheduler with an associated pool of messages. Each chare is assumed to reside on one PE at a time. A logical node is implemented as an OS process. In non-SMP mode there is no distinction between a PE and a logical node. Otherwise, a PE takes the form of an OS thread, and a logical node may contain one or more PEs. Physical nodes may be partitioned into one or more logical nodes. Since PEs within a logical node share the same memory address space, the Charm++ runtime system optimizes communication between them by using shared memory. Depending on the runtime command-line parameters, a PE may optionally be associated with a subset of cores or hardware threads.

A Charm++ program can be launched with one or more (logical) nodes per physical node. For example, on a machine with a four-core processor, where each core has two hardware threads, common configurations in non-SMP mode would be one node per core (four nodes/PEs total) or one node per hardware thread (eight nodes/PEs total). In SMP

² For a description of the underlying design philosophy please refer to the following papers:

- L. V. Kale and Sanjeev Krishnan, “Charm++: *Parallel Programming with Message-Driven Objects*”, in “Parallel Programming Using C++”, MIT Press, 1995.
- L. V. Kale and Sanjeev Krishnan, “Charm++: *A Portable Concurrent Object Oriented System Based On C++*”, Proceedings of the Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), September 1993.

mode, the most common choice to fully subscribe the physical node would be one logical node containing *seven* PEs—one OS thread is set aside per process for network communications. (When built in the “multicore” mode that lacks network support, a comm thread is unnecessary, and eight PEs can be used in this case. A comm thread is also omitted when using some high-performance network layers such as PAMI.) Alternatively, one can choose to partition the physical node into multiple logical nodes, each containing multiple PEs. One example would be *three* PEs per logical node and two logical nodes per physical node, again reserving a comm thread per logical node.

It is not a general practice in Charm++ to oversubscribe the underlying physical cores or hardware threads on each node. In other words, a Charm++ program is usually not launched with more PEs than there are physical cores or hardware threads allocated to it. More information about these launch time options are provided in Appendix 2.6.3. And utility functions to retrieve the information about those Charm++ logical machine entities in user programs can be referred in section 2.2.1.

2.2 Basic Charm++ Programming

2.2.1 Program Structure, Compilation and Utilities

A Charm++ program is essentially a C++ program where some components describe its parallel structure. Sequential code can be written using any programming technologies that cooperate with the C++ toolchain. This includes C and Fortran. Parallel entities in the user’s code are written in C++. These entities interact with the Charm++ framework via inherited classes and function calls.

Charm++ Interface (.ci) Files

All user program components that comprise its parallel interface (such as messages, chares, entry methods, etc.) are granted this elevated status by declaring them in separate *charm++ interface* description files. These files have a *.ci* suffix and adopt a C++-like declaration syntax with several additional keywords. In some declaration contexts, they may also contain some sequential C++ source code. Charm++ parses these interface descriptions and generates C++ code (base classes, utility classes, wrapper functions etc.) that facilitates the interaction of the user program’s entities with the framework. A program may have several interface description files.

Syntax Highlighting of .ci Files

Vim

To enable syntax highlighting of .ci files in Vim, do the following:

```
$ cp charm/contrib/ci.vim ~/.vim/syntax/.  
$ vim ~/.vim/filetype.vim
```

And paste the following line in that file:

```
au! BufRead,BufNewFile *.ci set filetype=ci
```

Sublime Text

Syntax highlighting in Sublime Text (version 3 or newer) can be enabled by installing the *Charmci* package through Package Control.

Emacs

Syntax highlighting in Emacs can be enabled by triggering C++ handling on the `.ci` file extension by adding the following line to your `.emacs` file.

```
(add-to-list 'auto-mode-alist '("\\.ci\\'" . c++-mode))
```

Pygments

Pygments supports syntax highlighting of `.ci` files starting with version 2.4.0, when setting `charmci` as the highlighting language, or automatically for files with the `.ci` filename extension.

Modules

The top-level construct in a *ci* file is a named container for interface declarations called a *module*. Modules allow related declarations to be grouped together, and cause generated code for these declarations to be grouped into files named after the module. Modules cannot be nested, but each *ci* file can have several modules. Modules are specified using the keyword *module*. A module name must be a valid C++ identifier.

```
module myFirstModule {
    // Parallel interface declarations go here
    ...
};
```

Generated Files

Each module present in a *ci* file is parsed to generate two files. The basename of these files is the same as the name of the module and their suffixes are `.decl.h` and `.def.h`. For e.g., the module defined earlier will produce the files “myFirstModule.decl.h” and “myFirstModule.def.h”. As the suffixes indicate, they contain the declarations and definitions respectively, of all the classes and functions that are generated based on the parallel interface description.

We recommend that the header file containing the declarations (`decl.h`) be included at the top of the files that contain the declarations or definitions of the user program entities mentioned in the corresponding module. The `def.h` is not actually a header file because it contains definitions for the generated entities. To avoid multiple definition errors, it should be compiled into just one object file. A convention we find useful is to place the `def.h` file at the bottom of the source file (`.C`, `.cpp`, `.cc` etc.) which includes the definitions of the corresponding user program entities.

It should be noted that the generated files have no dependence on the name of the *ci* file, but only on the names of the modules. This can make automated dependency-based build systems slightly more complicated.

Module Dependencies

A module may depend on the parallel entities declared in another module. It can express this dependency using the *extern* keyword. *extern* ed modules do not have to be present in the same *ci* file.

```
module mySecondModule {
    // Entities in this module depend on those declared in another module
    extern module myFirstModule;

    // More parallel interface declarations
```

(continues on next page)

(continued from previous page)

```
    ...  
};
```

The *extern* keyword places an include statement for the *decl.h* file of the *extern* ed module in the generated code of the current module. Hence, *decl.h* files generated from *extern* ed modules are required during the compilation of the source code for the current module. This is usually required anyway because of the dependencies between user program entities across the two modules.

The Main Module and Reachable Modules

Charm++ software can contain several module definitions from several independently developed libraries / components. However, the user program must specify exactly one module as containing the starting point of the program's execution. This module is called the *mainmodule*. Every Charm++ program has to contain precisely one *mainmodule*.

All modules that are “reachable” from the *mainmodule* via a chain of *extern* ed module dependencies are included in a Charm++ program. More precisely, during program execution, the Charm++ runtime system will recognize only the user program entities that are declared in reachable modules. The *decl.h* and *def.h* files may be generated for other modules, but the runtime system is not aware of entities declared in such unreachable modules.

```
module A {  
    ...  
};  
  
module B {  
    extern module A;  
    ...  
};  
  
module C {  
    extern module A;  
    ...  
};  
  
module D {  
    extern module B;  
    ...  
};  
  
module E {  
    ...  
};  
  
mainmodule M {  
    extern module C;  
    extern module D;  
    // Only modules A, B, C and D are reachable and known to the runtime system  
    // Module E is unreachable via any chain of externed modules  
    ...  
};
```

Including other headers

There can be occasions where code generated from the module definitions requires other declarations / definitions in the user program's sequential code. Usually, this can be achieved by placing such user code before the point of

inclusion of the decl.h file. However, this can become laborious if the decl.h file has to be included in several places. Charm++ supports the keyword *include* in *ci* files to permit the inclusion of any header directly into the generated decl.h files.

```
module A {
    include "myUtilityClass.h"; //< Note the semicolon
    // Interface declarations that depend on myUtilityClass
    ...
};

module B {
    include "someUserTypedefs.h";
    // Interface declarations that require user typedefs
    ...
};

module C {
    extern module A;
    extern module B;
    // The user includes will be indirectly visible here too
    ...
};
```

The main() function

The Charm++ framework implements its own main function and retains control until the parallel execution environment is initialized and ready for executing user code. Hence, the user program must not define a *main()* function. Control enters the user code via the *mainchare* of the *mainmodule*. This will be discussed in further detail in 2.1.2.

Using the facilities described thus far, the parallel interface declarations for a Charm++ program can be spread across multiple *ci* files and multiple modules, permitting good control over the grouping and export of parallel API. This aids the encapsulation of parallel software.

Compiling Charm++ Programs

Charm++ provides a compiler-wrapper called *charmcc* that handles all *ci*, C, C++ and Fortran source files that are part of a user program. Users can invoke *charmcc* to parse their interface descriptions, compile source code and link objects into binaries. It also links against the appropriate set of charm framework objects and libraries while producing a binary. *charmcc* and its functionality is described in 2.6.2.

Utility Functions

The following calls provide basic rank information and utilities useful when running a Charm++ program.

`void CkAssert(int expression)` Aborts the program if expression is 0.

`void CkAbort(const char *format, ...)` Causes the program to abort, printing the given error message. Supports printf-style formatting. This function never returns.

`void CkExit()` This call informs the Charm RTS that computation on all processors should terminate. This routine never returns, so any code after the call to `CkExit()` inside the function that calls it will not execute. Other processors will continue executing until they receive notification to stop, so it is a good idea to ensure through synchronization that all useful work has finished before calling `CkExit()`.

`double CkWallTimer()` Returns the elapsed wall time since the start of execution in seconds.

Information about Logical Machine Entities

As described in section 2.1.4, Charm++ recognizes two logical machine entities: “node” and PE (processing element). The following functions provide basic information about such logical machine that a Charm++ program runs on. PE and “node” are numbered starting from zero.

`int CkNumPes()` Returns the total number of PEs across all nodes.

`int CkMyPe()` Returns the index of the PE on which the call was made.

`int CkNumNodes()` Returns the total number of logical Charm++ nodes.

`int CkMyNode()` Returns the index of the “node” on which the call was made.

`int CkMyRank()` Returns the rank number of the PE on a “node” on which the call was made. PEs within a “node” are also ranked starting from zero.

`int CkNodeFirst(int nd)` Returns the index of the first PE on the logical node *nd*.

`int CkNodeSize(int nd)` Returns the number of PEs on the logical node *nd* on which the call was made.

`int CkNodeOf(int pe)` Returns the “node” number that PE *pe* belongs to.

`int CkRankOf(int pe)` Returns the rank of the given PE within its node.

Terminal I/O

Charm++ provides both C and C++ style methods of doing terminal I/O.

In place of C-style `printf` and `scanf`, Charm++ provides `CkPrintf` and `CkScanf`. These functions have interfaces that are identical to their C counterparts, but there are some differences in their behavior that should be mentioned.

Charm++ also supports all forms of `printf`, `cout`, etc. in addition to the special forms shown below. The special forms below are still useful, however, since they obey well-defined (but still lax) ordering requirements.

`int CkPrintf(format [, arg]*)` This call is used for atomic terminal output. Its usage is similar to `printf` in C. However, `CkPrintf` has some special properties that make it more suited for parallel programming. `CkPrintf` routes all terminal output to a single end point which prints the output. This guarantees that the output for a single call to `CkPrintf` will be printed completely without being interleaved with other calls to `CkPrintf`. Note that `CkPrintf` is implemented using an asynchronous send, meaning that the call to `CkPrintf` returns immediately after the message has been sent, and most likely before the message has actually been received, processed, and displayed. As such, there is no guarantee of order in which the output for concurrent calls to `CkPrintf` is printed. Imposing such an order requires proper synchronization between the calls to `CkPrintf` in the parallel application.

`void CkError(format [, arg]*)` Like `CkPrintf`, but used to print error messages on `stderr`.

`int CkScanf(format [, arg]*)` This call is used for atomic terminal input. Its usage is similar to `scanf` in C. A call to `CkScanf`, unlike `CkPrintf`, blocks all execution on the processor it is called from, and returns only after all input has been retrieved.

For C++ style stream-based I/O, Charm++ offers `ckout` and `ckerr` in place of `cout` and `cerr`. The C++ streams and their Charm++ equivalents are related in the same manner as `printf` and `scanf` are to `CkPrintf` and `CkScanf`. The Charm++ streams are all used through the same interface as the C++ streams, and all behave in a slightly different way, just like C-style I/O.

2.2.2 Basic Syntax

Entry Methods

Member functions in the user program which function as entry methods have to be defined in public scope within the class definition. Entry methods typically do not return data and have a “void” return type. An entry method with the same name as its enclosing class is a constructor entry method and is used to create or spawn chare objects during execution. Class member functions are annotated as entry methods by declaring them in the interface file as:

```
entry void Entry1(parameters);
```

Parameters is either a list of serializable parameters, (e.g., “int i, double x”), or a message type (e.g., “MyMessage *msg”). Since parameters get marshalled into a message before being sent across the network, in this manual we use “message” to mean either a message type or a set of marshalled parameters.

Messages are lower level, more efficient, more flexible to use than parameter marshalling.

For example, a chare could have this entry method declaration in the interface (.ci) file:

```
entry void foo(int i, int k);
```

Then invoking foo(2,3) on the chare proxy will eventually invoke foo(2,3) on the chare object.

Since Charm++ runs on distributed memory machines, we cannot pass an array via a pointer in the usual C++ way. Instead, we must specify the length of the array in the interface file, as:

```
entry void bar(int n, double arr[n]);
```

Since C++ does not recognize this syntax, the array data must be passed to the chare proxy as a simple pointer. The array data will be copied and sent to the destination processor, where the chare will receive the copy via a simple pointer again. The remote copy of the data will be kept until the remote method returns, when it will be freed. This means any modifications made locally after the call will not be seen by the remote chare; and the remote chare’s modifications will be lost after the remote method returns- Charm++ always uses call-by-value, even for arrays and structures.

This also means the data must be copied on the sending side, and to be kept must be copied again at the receive side. Especially for large arrays, this is less efficient than messages, as described in the next section.

Array parameters and other parameters can be combined in arbitrary ways, as:

```
entry void doLine(float data[n], int n);
entry void doPlane(float data[n*n], int n);
entry void doSpace(int n, int m, int o, float data[n*m*o]);
entry void doGeneral(int nd, int dims[nd], float data[product(dims, nd)]);
```

The array length expression between the square brackets can be any valid C++ expression, including a fixed constant, and may depend in any manner on any of the passed parameters or even on global functions or global data. The array length expression is evaluated exactly once per invocation, on the sending side only. Thus executing the doGeneral method above will invoke the (user-defined) product function exactly once on the sending processor.

Marshalling User-Defined Structures and Classes

The marshalling system uses the pup framework to copy data, meaning every user class that is marshalled needs either a pup routine, a “PUPbytes” declaration, or a working operator!. See the PUP description in Section 2.2.5 for more details on these routines.

Any user-defined types in the argument list must be declared before including the “decl.h” file. Any user-defined types must be fully defined before the entry method declaration that consumes it. This is typically done by including

the header defining the type in the `.ci` file. Alternatively, one may define it before including the `.decl.h` file. As usual in C, it is often dramatically more efficient to pass a large structure by reference than by value.

As an example, refer to the following code from `examples/charm++/PUP/HeapPUP`:

```
// In HeapObject.h:

class HeapObject {
public:
    int publicInt;

    // ... other methods ...

    void pup(PUP::er &p) {
        // remember to pup your superclass if there is one
        p|publicInt;
        p|privateBool;
        if (p.isUnpacking())
            data = new float[publicInt];
        PUPArray(p, data, publicInt);
    }

private:
    bool privateBool;
    float *data;
};

// In SimplePup.ci:

mainmodule SimplePUP {
    include "HeapObject.h";

    // ... other Chare declarations ...

    array [1D] SimpleArray{
        entry SimpleArray();
        entry void acceptData(HeapObject &inData);
    };
};

// In SimplePup.h:

#include "SimplePUP.decl.h"

// ... other definitions ...

class SimpleArray : public CBase_SimpleArray {
public:
    void acceptData(HeapObject &inData) {
        // ... code using marshalled parameter ...
    }
};

// In SimplePup.C:

#include "SimplePUP.h"

main::main(CkArgMsg *m)
```

(continues on next page)

(continued from previous page)

```

{
    // normal object construction
    HeapObject exampleObject(... parameters ...);

    // normal chare array construction
    CProxy_SimpleArray simpleProxy = CProxy_SimpleArray::ckNew(30);

    // pass object to remote method invocation on the chare array
    simpleProxy[29].acceptData(exampleObject);
}

#include "SimplePUP.def.h"

```

Chare Objects

Chares are concurrent objects with methods that can be invoked remotely. These methods are known as entry methods. All chares must have a constructor that is an entry method, and may have any number of other entry methods. All chare classes and their entry methods are declared in the interface (.ci) file:

```

chare ChareType
{
    entry ChareType(parameters1);
    entry void EntryMethodName(parameters2);
};

```

Although it is *declared* in an interface file, a chare is a C++ object and must have a normal C++ *implementation* (definition) in addition. A chare class ChareType must inherit from the class CBase_ChareType, which is a special class that is generated by the Charm++ translator from the interface file. Note that C++ namespace constructs can be used in the interface file, as demonstrated in `examples/charm++/namespace`.

To be concrete, the C++ definition of the chare above might have the following definition in a .h file:

```

class ChareType : public CBase_ChareType {
    // Data and member functions as in C++
public:
    ChareType(parameters1);
    void EntryMethodName2(parameters2);
};

```

Each chare encapsulates data associated with medium-grained units of work in a parallel application. Chares can be dynamically created on any processor; there may be thousands of chares on a processor. The location of a chare is usually determined by the dynamic load balancing strategy. However, once a chare commences execution on a processor, it does not migrate to other processors³. Chares do not have a default “thread of control”: the entry methods in a chare execute in a message driven fashion upon the arrival of a message⁴.

The entry method definition specifies a function that is executed *without interruption* when a message is received and scheduled for processing. Only one message per chare is processed at a time. Entry methods are defined exactly as normal C++ function members, except that they must have the return value void (except for the constructor entry method which may not have a return value, and for a *synchronous* entry method, which is invoked by a *threaded* method in a remote chare). Each entry method can either take no arguments, take a list of arguments that the runtime system can automatically pack into a message and send (see section 2.2.2), or take a single argument that is a pointer to a Charm++ message (see section 2.3.1).

³ Except when it is part of an array.

⁴ Threaded methods augment this behavior since they execute in a separate user-level thread, and thus can block to wait for data.

A chare's entry methods can be invoked via *proxies* (see section 2.1.3). Proxies to a chare of type `chareType` have type `CProxy_chareType`. By inheriting from the `CBase` parent class, each chare gets a `thisProxy` member variable, which holds a proxy to itself. This proxy can be sent to other chares, allowing them to invoke entry methods on this chare.

Chare Creation

Once you have declared and defined a chare class, you will want to create some chare objects to use. Chares are created by the `ckNew` method, which is a static method of the chare's proxy class:

```
CProxy_chareType::ckNew(parameters, int destPE);
```

The `parameters` correspond to the parameters of the chare's constructor. Even if the constructor takes several arguments, all of the arguments should be passed in order to `ckNew`. If the constructor takes no arguments, the `parameters` are omitted. By default, the new chare's location is determined by the runtime system. However, this can be overridden by passing a value for `destPE`, which specifies the PE where the chare will be created.

The chare creation method deposits the *seed* for a chare in a pool of seeds and returns immediately. The chare will be created later on some processor, as determined by the dynamic load balancing strategy (or by `destPE`). When a chare is created, it is initialized by calling its constructor entry method with the parameters specified by `ckNew`.

Suppose we have declared a chare class `C` with a constructor that takes two arguments, an `int` and a `double`.

1. This will create a new chare of type `C` on any processor and return a proxy to that chare:

```
CProxy_C chareProxy = CProxy_C::ckNew(1, 10.0);
```

2. This will create a new chare of type `C` on processor `destPE` and return a proxy to that chare:

```
CProxy_C chareProxy = CProxy_C::ckNew(1, 10.0, destPE);
```

For an example of chare creation in a full application, see `examples/charm++/fib` in the Charm++ software distribution, which calculates Fibonacci numbers in parallel.

Method Invocation on Chares

A message may be sent to a chare through a proxy object using the notation:

```
chareProxy.EntryMethod(parameters)
```

This invokes the entry method `EntryMethod` on the chare referred to by the proxy `chareProxy`. This call is asynchronous and non-blocking; it returns immediately after sending the message.

Local Access

You can get direct access to a local chare using the proxy's `ckLocal` method, which returns an ordinary C++ pointer to the chare if it exists on the local processor, and `NULL` otherwise.

```
C *c=chareProxy.ckLocal();  
if (c==NULL) {  
    // object is remote; send message  
}  
else {  
    // object is local; directly use members and methods of c  
}
```


Read-only Data

Since Charm++ does not allow global variables, it provides a special mechanism for sharing data amongst all objects. *Read-only* variables of simple data types or compound data types including messages and arrays are used to share information that is obtained only after the program begins execution and does not change after they are initialized in the dynamic scope of the `main` function of the `mainchare`. They are broadcast to every Charm++ Node (process) by the Charm++ runtime, and can be accessed in the same way as C++ “global” variables on any process. Compound data structures containing pointers can be made available as read-only variables using read-only messages (see section 2.3.1) or read-only arrays (see section 2.2.3). Note that memory has to be allocated for read-only messages by using `new` to create the message in the `main` function of the `mainchare`.

Read-only variables are declared by using the type modifier `readonly`, which is similar to `const` in C++. Read-only data is specified in the `.ci` file (the interface file) as:

```
readonly Type ReadonlyVarName;
```

The variable `ReadonlyVarName` is declared to be a read-only variable of type `Type`. `Type` must be a single token and not a type expression.

```
readonly message MessageType *ReadonlyMsgName;
```

The variable `ReadonlyMsgName` is declared to be a read-only message of type `MessageType`. Pointers are not allowed to be `readonly` variables unless they are pointers to message types. In this case, the message will be initialized on every PE.

```
readonly Type ReadonlyArrayName [arraysize];
```

The variable `ReadonlyArrayName` is declared to be a read-only array of type `Type` with `arraysize` elements. `Type` must be a single token and not a type expression. The value of `arraysize` must be known at compile time.

Read-only variables must be declared either as global or as public class static data in the C/C++ implementation files, and these declarations have the usual form:

```
Type ReadonlyVarName;
MessageType *ReadonlyMsgName;
Type ReadonlyArrayName [arraysize];
```

Similar declarations preceded by `extern` would appear in the `.h` file.

Note: The current Charm++ translator cannot prevent assignments to read-only variables. The user must make sure that no assignments occur in the program outside of the `mainchare` constructor.

For concrete examples for using read-only variables, please refer to examples such as `examples/charm++/array` and `examples/charm++/gaussSeidel3D`.

Users can get the same functionality of `readonly` variables by making such variables members of Charm++ Node Group objects and constructing the Node Group in the `mainchare`’s main routine.

2.2.3 Chare Arrays

Chare arrays are arbitrarily-sized, possibly-sparse collections of chares that are distributed across the processors. The entire array has a globally unique identifier of type `CkArrayID`, and each element has a unique index of type `CkArrayIndex`. A `CkArrayIndex` can be a single integer (i.e. a one-dimensional array), several integers (i.e. a multi-dimensional array), or an arbitrary string of bytes (e.g. a binary tree index).

Array elements can be dynamically created and destroyed on any PE, migrated between PEs, and messages for the elements will still arrive properly. Array elements can be migrated at any time, allowing arrays to be efficiently load balanced. A chare array (or a subset of array elements) can receive a broadcast/multicast or contribute to a reduction.

An example program can be found here: `examples/charm++/array`.

Declaring a One-dimensional Array

You can declare a one-dimensional (1D) chare array as:

```
//In the .ci file:
array [1D] A {
    entry A(parameters1);
    entry void someEntry(parameters2);
};
```

Array elements extend the system class `CBase_ClassName`, inheriting several fields:

- `thisProxy`: the proxy to the entire chare array that can be indexed to obtain a proxy to a specific array element (e.g. for a 1D chare array `thisProxy[10]`; for a 2D chare array `thisProxy(10, 20)`)
- `thisArrayID`: the array's globally unique identifier
- `thisIndex`: the element's array index (an array element can obtain a proxy to itself like this `thisProxy[thisIndex]`)

```
class A : public CBase_A {
public:
    A(parameters1);

    void someEntry(parameters2);
};
```

Note that `A` must have a *migration constructor* if it is to be migratable. The migration constructor is typically empty:

```
//In the .C file:
A::A(void)
{
    //... constructor code ...
}

A::someEntry(parameters2)
{
    // ... code for someEntry ...
}
```

See the section 2.2.5 on migratable array elements for more information on the migration constructor that takes `Ck-MigrateMessage *` as the argument.

Declaring Multi-dimensional Arrays

Charm++ supports multi-dimensional or user-defined indices. These array types can be declared as:

```
//In the .ci file:
array [1D] ArrayA { entry ArrayA(); entry void e(parameters); }
array [2D] ArrayB { entry ArrayB(); entry void e(parameters); }
array [3D] ArrayC { entry ArrayC(); entry void e(parameters); }
array [4D] ArrayD { entry ArrayD(); entry void e(parameters); }
array [5D] ArrayE { entry ArrayE(); entry void e(parameters); }
array [6D] ArrayF { entry ArrayF(); entry void e(parameters); }
array [Foo] ArrayG { entry ArrayG(); entry void e(parameters); }
array [Bar<3>] ArrayH { entry ArrayH(); entry void e(parameters); }
```

The declaration of `ArrayG` expects an array index of type `CkArrayIndexFoo`, which must be defined before including the `.decl.h` file (see section 2.3.4 on user-defined array indices for more information).

```
//In the .h file:
class ArrayA : public CBase_ArrayA { public: ArrayA() {} ...};
class ArrayB : public CBase_ArrayB { public: ArrayB() {} ...};
class ArrayC : public CBase_ArrayC { public: ArrayC() {} ...};
class ArrayD : public CBase_ArrayD { public: ArrayD() {} ...};
class ArrayE : public CBase_ArrayE { public: ArrayE() {} ...};
class ArrayF : public CBase_ArrayF { public: ArrayF() {} ...};
class ArrayG : public CBase_ArrayG { public: ArrayG() {} ...};
class ArrayH : public CBase_ArrayH { public: ArrayH() {} ...};
```

The fields in `thisIndex` are different depending on the dimensionality of the chare array:

- 1D array: `thisIndex`
- 2D array (x,y): `thisIndex.x`, `thisIndex.y`
- 3D array (x,y,z): `thisIndex.x`, `thisIndex.y`, `thisIndex.z`
- 4D array (w,x,y,z): `thisIndex.w`, `thisIndex.x`, `thisIndex.y`, `thisIndex.z`
- 5D array (v,w,x,y,z): `thisIndex.v`, `thisIndex.w`, `thisIndex.x`, `thisIndex.y`, `thisIndex.z`
- 6D array (x_1,y_1,z_1,x_2,y_2,z_2): `thisIndex.x1`, `thisIndex.y1`, `thisIndex.z1`, `thisIndex.x2`, `thisIndex.y2`, `thisIndex.z2`
- Foo array: `thisIndex`
- `Bar<3>` array: `thisIndex`

Creating an Array

An array is created using the `CProxy_Array::ckNew` routine, which must be called from PE 0. To create an array from any PE, asynchronous array creation using a callback can be used. See section 2.3.4 for asynchronous array creation. `CProxy_Array::ckNew` returns a proxy object, which can be kept, copied, or sent in messages. The following creates a 1D array containing elements indexed (0, 1, ..., `dimX-1`):

```
CProxy_ArrayA a1 = CProxy_ArrayA::ckNew(params, dimX);
```

Likewise, a dense multidimensional array can be created by passing the extents at creation time to `ckNew`.

```
CProxy_ArrayB a2 = CProxy_ArrayB::ckNew(params, dimX, dimY);
CProxy_ArrayC a3 = CProxy_ArrayC::ckNew(params, dimX, dimY, dimZ);
CProxy_ArrayD a4 = CProxy_ArrayC::ckNew(params, dimW, dimX, dimY, dimZ);
CProxy_ArrayE a5 = CProxy_ArrayC::ckNew(params, dimV, dimW, dimX, dimY, dimZ);
CProxy_ArrayF a6 = CProxy_ArrayC::ckNew(params, dimX1, dimY1, dimZ1, dimX2, dimY2,
↪dimZ2);
```

For user-defined arrays, this functionality cannot be used. The array elements must be inserted individually as described in section 2.3.4.

During creation, the constructor is invoked on each array element. For more options when creating the array, see section 2.3.4.

Entry Method Invocation

To obtain a proxy to a specific element in chare array, the chare array proxy (e.g. `thisProxy`) must be indexed by the appropriate index call depending on the dimensionality of the array:

- 1D array, to obtain a proxy to element i : `thisProxy[i]` or `thisProxy(i)`
- 2D array, to obtain a proxy to element (i, j) : `thisProxy(i, j)`
- 3D array, to obtain a proxy to element (i, j, k) : `thisProxy(i, j, k)`
- 4D array, to obtain a proxy to element (i, j, k, l) : `thisProxy(i, j, k, l)`
- 5D array, to obtain a proxy to element (i, j, k, l, m) : `thisProxy(i, j, k, l, m)`
- 6D array, to obtain a proxy to element (i, j, k, l, m, n) : `thisProxy(i, j, k, l, m, n)`
- User-defined array, to obtain a proxy to element i : `thisProxy[i]` or `thisProxy(i)`

To send a message to an array element, index the proxy and call the method name:

```
a1[i].doSomething(parameters);  
a3(x, y, z).doAnother(parameters);  
aF[CkArrayIndexFoo(...)].doAgain(parameters);
```

You may invoke methods on array elements that have not yet been created. The Charm++ runtime system will buffer the message until the element is created.⁵

Messages are not guaranteed to be delivered in order. For instance, if a method is invoked on method A and then method B; it is possible that B is executed before A.

```
a1[i].A();  
a1[i].B();
```

Messages sent to migrating elements will be delivered after the migrating element arrives on the destination PE. It is an error to send a message to a deleted array element.

Broadcasts on Chare Arrays

To broadcast a message to all the current elements of an array, simply omit the index (invoke an entry method on the chare array proxy):

```
a1.doIt(parameters); //<- invokes doIt on each array element
```

The broadcast message will be delivered to every existing array element exactly once. Broadcasts work properly even with ongoing migrations, insertions, and deletions.

Reductions on Chare Arrays

A reduction applies a single operation (e.g. add, max, min, ...) to data items scattered across many processors and collects the result in one place. Charm++ supports reductions over the members of an array or group.

The data to be reduced comes from a call to the member contribute method:

```
void contribute(int nBytes, const void *data, CkReduction::reducerType type);
```

This call contributes `nBytes` bytes starting at `data` to the reduction type (see Section 2.2.3). Unlike sending a message, you may use `data` after the call to `contribute`. All members of the chare array or group must call `contribute`, and all of them must use the same reduction type.

For example, if we want to sum each array/group member's single integer `myInt`, we would use:

⁵ However, the element must eventually be created.

```
// Inside any member method
int myInt=get_myInt();
contribute(sizeof(int), &myInt, CkReduction::sum_int);
```

The built-in reduction types (see below) can also handle arrays of numbers. For example, if each element of a chare array has a pair of doubles `forces[2]`, the corresponding elements of which are to be added across all elements, from each element call:

```
double forces[2]=get_my_forces();
contribute(2*sizeof(double), forces, CkReduction::sum_double);
```

Note that since C++ arrays (like `forces[2]`) are already pointers, we don't use `&forces`.

A slightly simpler interface is available for `std::vector<T>`, since the class determines the size and count of the underlying type:

```
CkCallback cb(...);
vector<double> forces(2);
get_my_forces(forces);
contribute(forces, CkReduction::sum_double, cb);
```

Either of these will result in a double array of 2 elements, the first of which contains the sum of all `forces[0]` values, with the second element holding the sum of all `forces[1]` values of the chare array elements.

Typically the client entry method of a reduction takes a single argument of type `CkReductionMsg` (see Section 2.3.8). However, by giving an entry method the `reductiontarget` attribute in the `.ci` file, you can instead use entry methods that take arguments of the same type as specified by the `contribute` call. When creating a callback to the reduction target, the entry method index is generated by `CkReductionTarget(ChareClass, method_name)` instead of `CkIndex_ChareClass::method_name(...)`. For example, the code for a typed reduction that yields an `int`, would look like this:

```
// In the .ci file...
entry [reductiontarget] void done(int result);

// In some .C file:
// Create a callback that invokes the typed reduction client
// driverProxy is a proxy to the chare object on which
// the reduction target method "done" is called upon completion
// of the reduction
CkCallback cb(CkReductionTarget(Driver, done), driverProxy);

// Contribution to the reduction...
contribute(sizeof(int), &intData, CkReduction::sum_int, cb);

// Definition of the reduction client...
void Driver::done(int result)
{
    CkPrintf("Reduction value: %d", result);
}
```

This will also work for arrays of data elements (`entry [reductiontarget] void done(int n, int result[n])`), and for any user-defined type with a PUP method (see 2.2.5). If you know that the reduction will yield a particular number of elements, say 3 `ints`, you can also specify a reduction target which takes 3 `ints` and it will be invoked correctly.

Reductions do not have to specify commutative-associative operations on data; they can also be used to signal the fact that all array/group members have reached a certain synchronization point. In this case, a simpler version of `contribute` may be used:

```
contribute();
```

In all cases, the result of the reduction operation is passed to the *reduction client*. Many different kinds of reduction clients can be used, as explained in Section 2.3.8.

Please refer to `examples/charm++/reductions/typed_reduction` for a working example of reductions in Charm++.

Note that the reduction will complete properly even if char array elements are *migrated* or *deleted* during the reduction. Additionally, when you create a new char array element, it is expected to contribute to the next reduction not already in progress on that processor.

Built-in Reduction Types

Charm++ includes several built-in reduction types, used to combine individual contributions. Any of them may be passed as an argument of type `CkReduction::reducerType` to contribute.

The first four operations (`sum`, `product`, `max`, and `min`) work on `char`, `short`, `int`, `long`, `long long`, `float`, or `double` data as indicated by the suffix. The logical reductions (`and`, `or`) only work on `bool` and integer data. All the built-in reductions work on either single numbers (pass a pointer) or arrays- just pass the correct number of bytes to contribute.

1. `CkReduction::nop` : no operation performed.
2. `CkReduction::sum_char`, `sum_short`, `sum_int`, `sum_long`, `sum_long_long`, `sum_uchar`, `sum_ushort`, `sum_uint`, `sum_ulong`, `sum_ulong_long`, `sum_float`, `sum_double` : the result will be the sum of the given numbers.
3. `CkReduction::product_char`, `product_short`, `product_int`, `product_long`, `product_long_long`, `product_uchar`, `product_ushort`, `product_uint`, `product_ulong`, `product_ulong_long`, `product_float`, `product_double` : the result will be the product of the given numbers.
4. `CkReduction::max_char`, `max_short`, `max_int`, `max_long`, `max_long_long`, `max_uchar`, `max_ushort`, `max_uint`, `max_ulong`, `max_ulong_long`, `max_float`, `max_double` : the result will be the largest of the given numbers.
5. `CkReduction::min_char`, `min_short`, `min_int`, `min_long`, `min_long_long`, `min_uchar`, `min_ushort`, `min_uint`, `min_ulong`, `min_ulong_long`, `min_float`, `min_double` : the result will be the smallest of the given numbers.
6. `CkReduction::logical_and_bool`, `logical_and_int` : the result will be the logical AND of the given values.
7. `CkReduction::logical_or_bool`, `logical_or_int` : the result will be the logical OR of the given values.
8. `CkReduction::logical_xor_bool`, `logical_xor_int` : the result will be the logical XOR of the given values.
9. `CkReduction::bitvec_and_bool`, `bitvec_and_int` : the result will be the bitvector AND of the given values.
10. `CkReduction::bitvec_or_bool`, `bitvec_or_int` : the result will be the bitvector OR of the given values.
11. `CkReduction::bitvec_xor_bool`, `bitvec_xor_int` : the result will be the bitvector XOR of the given values.
12. `CkReduction::set` : the result will be a verbatim concatenation of all the contributed data, separated into `CkReduction::setElement` records. The data contributed can be of any length, and can vary across array elements or reductions. To extract the data from each element, see the description below.
13. `CkReduction::concat` : the result will be a byte-by-byte concatenation of all the contributed data. The contributed elements are not delimiter-separated.
14. `CkReduction::random` : the result will be a single randomly selected value of all of the contributed values.
15. `CkReduction::statistics` : returns a `CkReduction::statisticsElement` struct, containing summary statistics of the contributed data. Specifically, the struct contains the following fields: `int count`, `double mean`, and `double m2`, and the following functions: `double variance()` and `double stddev()`.

CkReduction::set returns a collection of CkReduction::setElement objects, one per contribution. This class has the definition:

```
class CkReduction::setElement
{
public:
    int dataSize; //The length of the 'data' array in bytes.
    char data[1]; //A place holder that marks the start of the data array.
    CkReduction::setElement *next(void);
};
```

Example: Suppose you would like to contribute 3 integers from each array element. In the reduction method you would do the following:

```
void ArrayClass::methodName (CkCallback &cb)
{
    int result[3];
    result[0] = 1;           // Copy the desired values into the result.
    result[1] = 2;
    result[2] = 3;
    // Contribute the result to the reductiontarget cb.
    contribute(3*sizeof(int), result, CkReduction::set, cb);
}
```

Inside the reduction's target method, the contributions can be accessed by using the CkReduction::setElement->next() iterator.

```
void SomeClass::reductionTargetMethod (CkReductionMsg *msg)
{
    // Get the initial element in the set.
    CkReduction::setElement *current = (CkReduction::setElement*) msg->getData();
    while(current != NULL) // Loop over elements in set.
    {
        // Get the pointer to the packed int's.
        int *result = (int*) &current->data;
        // Do something with result.
        current = current->next(); // Iterate.
    }
}
```

The reduction set order is undefined. You should add a source field to the contributed elements if you need to know which array element gave a particular contribution. Additionally, if the contributed elements are of a complex data type, you will likely have to supply code for serializing/deserializing them. Consider using the PUP interface (Section 2.2.5) to simplify your object serialization needs.

If the outcome of your reduction is dependent on the order in which data elements are processed, or if your data is just too heterogeneous to be handled elegantly by the predefined types and you don't want to undertake multiple reductions, you can use a tuple reduction or define your own custom reduction type.

Tuple reductions allow performing multiple different reductions in the same message. The reductions can be on the same or different data, and the reducer type for each reduction can be set independently as well. The contributions that make up a single tuple reduction message are all reduced in the same order as each other. As an example, a char array element can contribute to a gather-like operation using a tuple reduction that consists of two set reductions.

```
int tupleSize = 2;
CkReduction::tupleElement tupleRedn[] = {
    CkReduction::tupleElement(sizeof(int), &thisIndex, CkReduction::set),
    CkReduction::tupleElement(sizeData, data, CkReduction::set)
```

(continues on next page)

(continued from previous page)

```
};
CkReductionMsg* msg = CkReductionMsg::buildFromTuple(tupleRedn, tupleSize);
CkCallback allgatherCB(CkIndex_Foo::allgatherResult(0), thisProxy);
msg->setCallback(allgatherCB);
contribute(msg);
```

Note that `CkReduction::tupleElement` only holds pointers to the data that will make up the reduction message, therefore any local variables used must remain in scope until `CkReductionMsg::buildFromTuple` completes.

The result of this reduction is a single `CkReductionMsg` that can be processed as multiple reductions:

```
void Foo::allgatherResult (CkReductionMsg* msg)
{
    int numReductions;
    CkReduction::tupleElement* results;

    msg->toTuple(&results, &numReductions);
    CkReduction::setElement* currSrc = (CkReduction::setElement*)results[0].data;
    CkReduction::setElement* currData = (CkReduction::setElement*)results[1].data;

    // ... process currSrc and currData

    delete [] results;
}
```

See the next section (Section 2.3.8) for details on custom reduction types.

Destroying Array Elements

To destroy an array element - detach it from the array, call its destructor, and release its memory-invoke its `Array` `destroy` method, as:

```
a1[i].ckDestroy();
```

Note that this method can also be invoked remotely i.e. from a process different from the one on which the array element resides. You must ensure that no messages are sent to a deleted element. After destroying an element, you may insert a new element at its index.

2.2.4 Structured Control Flow: Structured Dagger

Charm++ is based on the message-driven parallel programming paradigm. In contrast to many other approaches, Charm++ programmers encode entry points to their parallel objects, but do not explicitly wait (i.e. block) on the runtime to indicate completion of posted ‘receive’ requests. Thus, a Charm++ object’s overall flow of control can end up fragmented across a number of separate methods, obscuring the sequence in which code is expected to execute. Furthermore, there are often constraints on when different pieces of code should execute relative to one another, related to data and synchronization dependencies.

Consider one way of expressing these constraints using flags, buffers, and counters, as in the following example:

```
// in .ci file
chare ComputeObject {
    entry void ComputeObject();
    entry void startStep();
```

(continues on next page)

(continued from previous page)

```

    entry void firstInput(Input i);
    entry void secondInput(Input j);
};

// in C++ file
class ComputeObject : public CBase_ComputeObject {
    int    expectedMessageCount;
    Input  first, second;

public:
    ComputeObject() {
        startStep();
    }
    void startStep() {
        expectedMessageCount = 2;
    }

    void firstInput(Input i) {
        first = i;
        if (--expectedMessageCount == 0)
            computeInteractions(first, second);
    }
    void recv_second(Input j) {
        second = j;
        if (--expectedMessageCount == 0)
            computeInteractions(first, second);
    }

    void computeInteractions(Input a, Input b) {
        // do computations using a and b
        ...
        // send off results
        ...
        // reset for next step
        startStep();
    }
};

```

In each step, this object expects pairs of messages, and waits to process the incoming data until it has both of them. This sequencing is encoded across 4 different functions, which in real code could be much larger and more numerous, resulting in a spaghetti-code mess.

Instead, it would be preferable to express this flow of control using structured constructs, such as loops. Charm++ provides such constructs for structured control flow across an object's entry methods in a notation called Structured Dagger. The basic constructs of Structured Dagger (SDAG) provide for *program-order execution* of the entry methods and code blocks that they define. These definitions appear in the `.ci` file definition of the enclosing `chare` class as a 'body' of an entry method following its signature.

The most basic construct in SDAG is the `serial` (aka the `atomic`) block. Serial blocks contain sequential C++ code. They're also called `atomic` because the code within them executes without returning control to the Charm++ runtime scheduler, and thus avoiding interruption from incoming messages. The keywords `atomic` and `serial` are synonymous, and you can find example programs that use `atomic`. However, we recommend the use of `serial` and are considering the deprecation of the `atomic` keyword. Typically serial blocks hold the code that actually deals with incoming messages in a `when` statement, or to do local operations before a message is sent or after it's received. The earlier example can be adapted to use serial blocks as follows:

```
// in .ci file
chare ComputeObject {
  entry void ComputeObject();
  entry void startStep();
  entry void firstInput(Input i) {
    serial {
      first = i;
      if (--expectedMessageCount == 0)
        computeInteractions(first, second);
    }
  };
  entry void secondInput(Input j) {
    serial {
      second = j;
      if (--expectedMessageCount == 0)
        computeInteractions(first, second);
    }
  };
};

// in C++ file
class ComputeObject : public CBase_ComputeObject {
  ComputeObject_SDAG_CODE
  int expectedMessageCount;
  Input first, second;

public:
  ComputeObject() {
    startStep();
  }
  void startStep() {
    expectedMessageCount = 2;
  }

  void computeInteractions(Input a, Input b) {
    // do computations using a and b
    . . .
    // send off results
    . . .
    // reset for next step
    startStep();
  }
};
```

Note that chare classes containing SDAG code must include a few additional declarations in addition to inheriting from their CBase_Foo class, by incorporating the Foo_SDAG_CODE generated-code macro in the class.

Serial blocks can also specify a textual ‘label’ that will appear in traces, as follows:

```
entry void firstInput(Input i) {
  serial "process first" {
    first = i;
    if (--expectedMessageCount == 0)
      computeInteractions(first, second);
  }
};
```

In order to control the sequence in which entry methods are processed, SDAG provides the `when` construct. These

statements, also called triggers, indicate that we expect an incoming message of a particular type, and provide code to handle that message when it arrives. From the perspective of a chare object reaching a `when` statement, it is effectively a ‘blocking receive.’

Entry methods defined by a `when` are not executed immediately when a message targeting them is delivered, but instead are held until control flow in the chare reaches a corresponding `when` clause. Conversely, when control flow reaches a `when` clause, the generated code checks whether a corresponding message has arrived: if one has arrived, it is processed; otherwise, control is returned to the Charm++ scheduler.

The use of `when` substantially simplifies the example from above:

```
// in .ci file
chare ComputeObject {
  entry void ComputeObject();
  entry void startStep() {
    when firstInput(Input first)
      when secondInput(Input second)
        serial {
          computeInteractions(first, second);
        }
  };
  entry void firstInput(Input i);
  entry void secondInput(Input j);
};

// in C++ file
class ComputeObject : public CBase_ComputeObject {
  ComputeObject_SDAG_CODE

public:
  ComputeObject() {
    startStep();
  }

  void computeInteractions(Input a, Input b) {
    // do computations using a and b
    . . .
    // send off results
    . . .
    // reset for next step
    startStep();
  }
};
```

Like an `if` or `while` in C code, each `when` clause has a body made up of the statement or block following it. The variables declared as arguments to the entry method triggering the `when` are available in the scope of the body. By using the sequenced execution of SDAG code and the availability of parameters to `when`-defined entry methods in their bodies, the counter `expectedMessageCount` and the intermediate copies of the received input are eliminated. Note that the entry methods `firstInput` and `secondInput` are still declared in the `.ci` file, but their definition is in the SDAG code. The interface translator generates code to handle buffering and triggering them appropriately.

For simplicity, `when` constructs can also specify multiple expected entry methods that all feed into a single body, by separating their prototypes with commas:

```
entry void startStep() {
  when firstInput(Input first),
    secondInput(Input second)
    serial {
```

(continues on next page)

(continued from previous page)

```

        computeInteractions(first, second);
    }
};

```

A single entry method is allowed to appear in more than one `when` statement. If only one of those `when` statements has been triggered when the runtime delivers a message to that entry method, that `when` statement is guaranteed to process it. If there is no trigger waiting for that entry method, then the next corresponding `when` to be reached will receive that message. If there is more than one `when` waiting on that method, which one will receive it is not specified, and should not be relied upon. For an example of multiple `when` statements handling the same entry method without reaching the unspecified case, see the CharmLU benchmark.

To more finely control the correspondence between incoming messages and `when` clauses associated with the target entry method, SDAG supports *matching* on reference numbers. Matching is typically used to denote an iteration of a program that executes asynchronously (without any sort of barrier or other synchronization between steps) or a particular piece of the problem being solved. Matching is requested by placing an expression denoting the desired reference number in square brackets between the entry method name and its parameter list. For parameter marshalled entry methods, the reference number expression will be compared for equality with the entry method's first argument. For entry methods that accept an explicit message (Section 2.3.1), the reference number on the message can be set by calling the function `CkSetRefNum(void *msg, CMK_REFNUM_TYPE ref)`. Matching is used in the loop example below, and in `examples/charm++/jacobi2d-sdag/jacobi2d.ci`. Multiple `when` triggers for an entry method with different matching reference numbers will not conflict - each will receive only corresponding messages.

SDAG supports the `for` and `while` loop constructs mostly as if they appeared in plain C or C++ code. In the running example, `computeInteractions()` calls `startStep()` when it is finished to start the next step. Instead of this arrangement, the loop structure can be made explicit:

```

// in .ci file
chare ComputeObject {
    entry void ComputeObject();
    entry void runForever() {
        while(true) {
            when firstInput(Input first),
                secondInput(Input second) serial {
                computeInteractions(first, second);
            }
        }
    };
    entry void firstInput(Input i);
    entry void secondInput(Input j);
};

// in C++ file
class ComputeObject : public CBase_ComputeObject {
    ComputeObject_SDAG_CODE

public:
    ComputeObject() {
        runForever();
    }

    void computeInteractions(Input a, Input b) {
        // do computations using a and b
        . . .
        // send off results
        . . .
    }
};

```

(continues on next page)

(continued from previous page)

```

    }
};

```

If this code should instead run for a fixed number of iterations, we can instead use a for loop:

```

// in .ci file
chare ComputeObject {
  entry void ComputeObject();
  entry void runForever() {
    for(iter = 0; iter < n; ++iter) {
      // Match to only accept inputs for the current iteration
      when firstInput[iter](int a, Input first),
        secondInput[iter](int b, Input second) serial {
        computeInteractions(first, second);
      }
    }
  };
  entry void firstInput(int a, Input i);
  entry void secondInput(int b, Input j);
};

// in C++ file
class ComputeObject : public CBase_ComputeObject {
  ComputeObject_SDAG_CODE
  int n, iter;

public:
  ComputeObject() {
    n = 10;
    runForever();
  }

  void computeInteractions(Input a, Input b) {
    // do computations using a and b
    . . .
    // send off results
    . . .
  }
};

```

Note that `int iter;` is declared in the chare's class definition and not in the `.ci` file. This is necessary because the Charm++ interface translator does not fully parse the declarations in the `for` loop header, because of the inherent complexities of C++. Finally, there is currently no mechanism by which to `break` or `continue` from an SDAG loop.

SDAG also supports conditional execution of statements and blocks with `if` statements. The syntax of SDAG `if` statements matches that of C and C++. However, if one encounters a syntax error on correct-looking code in a loop or conditional statement, try assigning the condition expression to a boolean variable in a serial block preceding the statement and then testing that boolean's value. This can be necessary because of the complexity of parsing C++ code.

In cases where multiple tasks must be processed before execution continues, but with no dependencies or interactions among them, SDAG provides the `overlap` construct. Overlap blocks contain a series of SDAG statements within them which can occur in any order. Commonly these blocks are used to hold a series of `when` triggers which can be received and processed in any order. Flow of control doesn't leave the overlap block until all the statements within it have been processed.

In the running example, suppose each input needs to be preprocessed independently before the call to

computeInteractions. Since we don't care which order they get processed in, and want it to happen as soon as possible, we can apply overlap:

```
// in .ci file
chare ComputeObject {
  entry void ComputeObject();
  entry void startStep() {
    overlap {
      when firstInput(Input i)
        serial { first = preprocess(i); }
      when secondInput(Input j)
        serial { second = preprocess(j); }
    }
    serial {
      computeInteractions(first, second);
    }
  };
  entry void firstInput(Input i);
  entry void secondInput(Input j);
};

// in C++ file
class ComputeObject : public CBase_ComputeObject {
  ComputeObject_SDAG_CODE

public:
  ComputeObject() {
    startStep();
  }

  void computeInteractions(Input a, Input b) {
    // do computations using a and b
    . . .
    // send off results
    . . .
    // reset for next step
    startStep();
  }
};
```

Another construct offered by SDAG is the `forall` loop. These loops are used when the iterations of a loop can be performed independently and in any order. This is in contrast to a regular `for` loop, in which each iteration is executed sequentially. The loop iterations are executed entirely on the calling PE, so they do not run in parallel. However, they are executed concurrently, in that execution of different iterations can interleave at `when` statements, like any other SDAG code. SDAG statements following the `forall` loop will not execute until all iterations have completed. The `forall` loop can be seen as an `overlap` with an indexed set of otherwise identical statements in the body.

The syntax of `forall` is

```
forall [IDENT] (MIN:MAX, STRIDE) BODY
```

The range from `MIN` to `MAX` is inclusive. In each iteration instance of `BODY`, the `IDENT` variable will take on one of the values in the specified range. The `IDENT` variable must be declared in the application C++ code as a member of the enclosing `chare` class.

Use of `forall` is demonstrated through distributed parallel matrix-matrix multiply shown in `examples/charm++/matmul/matmul.ci`

The case Statement

The `case` statement in SDAG expresses a disjunction over a set of `when` clauses. In other words, if it is known that one dependency out of a set will be satisfied, but which one is not known, this statement allows the set to be specified and will execute the corresponding block based on which dependency ends up being fulfilled.

The following is a basic example of the `case` statement. Note that the trigger `b()`, `d()` will only be fulfilled if both `b()` and `d()` arrive. If only one arrives, then it will partially match, and the runtime will not “commit” to this branch until the second arrives. If another dependency fully matches, the partial match will be ignored and can be used to trigger another `when` later in the execution.

```
case {
  when a() { }
  when b(), d() { }
  when c() { }
}
```

A full example of the `case` statement can be found `tests/charm++/sdag/case/caseTest.ci`.

Usage Notes

SDAG Code Declaration

If you’ve added *Structured Dagger* code to your class, you must link in the code by adding “`class-Name_SDAG_CODE`” inside the class declaration in the `.h` file. This macro defines the entry points and support code used by *Structured Dagger*. Forgetting this results in a compile error (undefined SDAG entry methods referenced from the `.def.h` file).

For example, an array named “Foo” that uses `sdag` code might contain:

```
class Foo : public CBase_Foo {
public:
  Foo_SDAG_CODE
  Foo(...) {
    ...
  }
  Foo(CkMigrateMessage *m) { }

  void pup(PUP::er &p) {
    ...
  }
  ...
};
```

Direct Calls to SDAG Entry Methods

An SDAG entry method that contains one or more `when` clause(s) cannot be directly called and will result in a runtime error with an error message. It has to be only called through a proxy. This is a runtime requirement that is enforced in order to prevent accidental calls to SDAG entry methods that are asynchronous in nature. Additionally, since they are called using a proxy, it enhances understandability and clarity as to not be confused for a regular function call that returns immediately.

For example, in the first example discussed, it is invalid to call the SDAG entry method `startStep` directly as `startStep()`; because it contains `when` clauses. It has to be only called using the proxy i.e. `computeObj.startStep()`; , where `computeObj` is the proxy to `ComputeObject`.

2.2.5 Serialization Using the PUP Framework

The PUP (Pack/Unpack) framework is a generic way to describe the data in an object and to use that description for serialization. The Charm++ system can use this description to pack the object into a message and unpack the message into a new object on another processor, to pack and unpack migratable objects for load balancing or checkpoint/restart-based fault tolerance. The PUP framework also includes support special for STL containers to ease development in C++.

Like many C++ concepts, the PUP framework is easier to use than describe:

```
class foo : public mySuperclass {
private:
    double a;
    int x;
    char y;
    unsigned long z;
    float arr[3];
public:
    ...other methods...

    //pack/unpack method: describe my fields to charm++
    void pup(PUP::er &p) {
        mySuperclass::pup(p);
        p|a;
        p|x; p|y; p|z;
        PUParray(p, arr, 3);
    }
};
```

This class's pup method describes the fields of the class to Charm++. This allows Charm++ to marshall parameters of type foo across processors, translate foo objects across processor architectures, read and write foo objects to files on disk, inspect and modify foo objects in the debugger, and checkpoint and restart programs involving foo objects.

PUP contract

Your object's pup method must save and restore all your object's data. As shown, you save and restore a class's contents by writing a method called "pup" which passes all the parts of the class to an object of type PUP::er, which does the saving or restoring. This manual will often use "pup" as a verb, meaning "to save/restore the value of" or equivalently, "to call the pup method of".

Pup methods for complicated objects normally call the pup methods for their simpler parts. Since all objects depend on their immediate superclass, the first line of every pup method is a call to the superclass's pup method—the only time you shouldn't call your superclass's pup method is when you don't have a superclass. If your superclass has no pup method, you must pup the values in the superclass yourself.

PUP operator

The recommended way to pup any object a is to use `p|a;`. This syntax is an operator `|` applied to the PUP::er `p` and the user variable `a`.

The `p|a;` syntax works wherever `a` is:

- A simple type, including char, short, int, long, float, or double. In this case, `p|a;` copies the data in-place. This is equivalent to passing the type directly to the PUP::er using `p(a)`.
- Any object with a pup method. In this case, `p|a;` calls the object's pup method. This is equivalent to the statement `a.pup(p);`.

- A pointer to a PUP::able object, as described in Section 2.3.9. In this case, `p|a;` allocates and copies the appropriate subclass.
- An object with a `PUPbytes(myClass)` declaration in the header. In this case, `p|a;` copies the object as plain bytes, like `memcpy`.
- An object with a custom operator `|` defined. In this case, `p|a;` calls the custom operator `|`.

See `examples/charm++/PUP`

For container types, you must simply `pup` each element of the container. For arrays, you can use the utility method `PUParray`, which takes the `PUP::er`, the array base pointer, and the array length. This utility method is defined for user-defined types `T` as:

```
template<class T>
inline void PUParray(PUP::er &p, T *array, int length) {
    for (int i=0; i<length; i++) p|array[i];
}
```

PUP STL Container Objects

If the variable is from the C++ Standard Template Library, you can include operator`|`'s for STL containers such as `vector`, `map`, `set`, `list`, `pair`, and `string`, templated on anything, by including the header "`pup_stl.h`".

See `examples/charm++/PUP/STLPUP`

PUP Dynamic Data

As usual in C++, pointers and allocatable objects usually require special handling. Typically this only requires a `p.isUnpacking()` conditional block, where you perform the appropriate allocation. See Section 2.3.9 for more information and examples.

If the object does not have a `pup` method, and you cannot add one or use `PUPbytes`, you can define an operator`|` to `pup` the object. For example, if `myClass` contains two fields `a` and `b`, the operator`|` might look like:

```
inline void operator| (PUP::er &p, myClass &c) {
    p|c.a;
    p|c.b;
}
```

See `examples/charm++/PUP/HeapPUP`

PUP as bytes

For classes and structs with many fields, it can be tedious and error-prone to list all the fields in the `pup` method. You can avoid this listing in two ways, as long as the object can be safely copied as raw bytes—this is normally the case for simple structs and classes without pointers.

- Use the `PUPbytes(myClass)` macro in your header file. This lets you use the `p|*myPtr;` syntax to `pup` the entire class as `sizeof(myClass)` raw bytes.
- Use `p((void *)myPtr, sizeof(myClass));` in the `pup` method. This is a direct call to `pup` a set of bytes.
- Use `p((char *)myCharArray, arraySize);` in the `pup` method. This is a direct call to `pup` a set of bytes. Other primitive types may also be used.

Note that pupping as bytes is just like using ‘memcpy’: it does nothing to the data other than copy it whole. For example, if the class contains any pointers, you must make sure to do any allocation needed, and pup the referenced data yourself.

Pupping as bytes may prevent your pup method from ever being able to work across different machine architectures. This is currently an uncommon scenario, but heterogeneous architectures may become more common, so pupping as bytes is discouraged.

PUP overhead

The PUP::er overhead is very small—one virtual function call for each item or array to be packed/unpacked. The actual packing/unpacking is normally a simple memory-to-memory binary copy.

For arrays and vectors of builtin arithmetic types like “int” and “double”, or of types declared as “PUPbytes”, PUParray uses an even faster block transfer, with one virtual function call per array or vector.

Thus, if an object does not contain pointers, you should prefer declaring it as PUPbytes.

For types of objects whose default constructors do more than necessary when an object will be unpacked from PUP, it is possible to tell the runtime system to call a more minimalistic alternative. This can apply to types used as both member variables of chares and as marshalled arguments to entry methods. A non-charge class can define a constructor that takes an argument of type `PUP::reconstruct` for this purpose. The runtime system code will call a `PUP::reconstruct` constructor in preference to a default constructor when it’s available. Where necessary, constructors taking `PUP::reconstruct` should call the constructors of members variables with `PUP::reconstruct` if applicable to that member.

PUP modes

Charm++ uses your pup method to both pack and unpack, by passing different types of PUP::ers to it. The method `p.isUnpacking()` returns true if your object is being unpacked—that is, your object’s values are being restored. Your pup method must work properly in sizing, packing, and unpacking modes; and to save and restore properly, the same fields must be passed to the PUP::er, in the exact same order, in all modes. This means most pup methods can ignore the pup mode.

Three modes are used, with three separate types of PUP::er: sizing, which only computes the size of your data without modifying it; packing, which reads/saves values out of your data; and unpacking, which writes/restores values into your data. You can determine exactly which type of PUP::er was passed to you using the `p.isSizing()`, `p.isPacking()`, and `p.isUnpacking()` methods. However, sizing and packing should almost always be handled identically, so most programs should use `p.isUnpacking()` and `!p.isUnpacking()`. Any program that calls `p.isPacking()` and does not also call `p.isSizing()` is probably buggy, because sizing and packing must see exactly the same data.

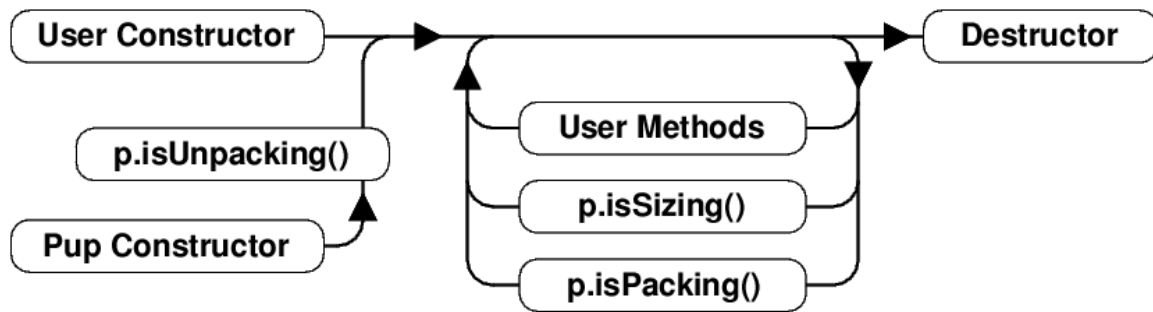
The `p.isDeleting()` flag indicates the object will be deleted after calling the pup method. This is normally only needed for pup methods called via the C or f90 interface, as provided by AMPI or the other frameworks. Other Charm++ array elements, marshalled parameters, and other C++ interface objects have their destructor called when they are deleted, so the `p.isDeleting()` call is not normally required—instead, memory should be deallocated in the destructor as usual.

Separately from indicating if the pup method is being used for sizing, packing, or unpacking, the system also provides methods to determine the purpose of a pup. The `p.isCheckpoint()` and `p.isMigration()` methods let the user determine if the runtime system has invoked the pup method for checkpointing (both in-memory and to disk) or for PE-to-PE migration (this is most commonly done for load balancing). These allow the user to customize their pup functions, for example, one may want to minimize checkpoint size by not including data that can be reconstructed. Note that these are orthogonal to the direction of the pup (e.g. `p.isCheckpoint()` will return true both when creating a checkpoint and when restoring from a checkpoint; one can differentiate between these two cases because

when creating a checkpoint `p.isPacking()` will also return true, while `p.isUnpacking()` will return true when restoring). The runtime system guarantees that at most of these will be set. There may be cases where neither `p.isCheckpoint()` nor `p.isMigration()` return true (e.g. when the system is marshalling entry method arguments).

More specialized modes and PUP::ers are described in section 2.3.9.

PUP Usage Sequence



1: Method sequence of an object with a pup method.

Typical method invocation sequence of an object with a pup method is shown in Figure 1. As usual in C++, objects are constructed, do some processing, and are then destroyed.

Objects can be created in one of two ways: they can be created using a normal constructor as usual; or they can be created using their pup constructor. The pup constructor for Charm++ array elements and PUP::able objects is a “migration constructor” that takes a single “CkMigrateMessage *” which the user should not free; for other objects, such as parameter marshalled objects, the pup constructor has no parameters. The pup constructor is always followed by a call to the object’s pup method in `isUnpacking` mode.

Once objects are created, they respond to regular user methods and remote entry methods as usual. At any time, the object pup method can be called in `isSizing` or `isPacking` mode. User methods and sizing or packing pup methods can be called repeatedly over the object lifetime.

Finally, objects are destroyed by calling their destructor as usual.

Migratable Array Elements using PUP

Array objects can migrate from one PE to another. For example, the load balancer (see section 2.2.6) might migrate array elements to better balance the load between processors. For an array element to be migratable, it must implement a pup method. The standard PUP contract (see section 2.2.5) and constraints wrt to serializing data apply. The one exception for `chare`, `group` and `node group` types is that since the runtime system will be the one to invoke their PUP routines, the runtime will automatically call PUP on the generated `CBase_` superclasses so users do not need to call PUP on generated superclasses.

A simple example for an array follows:

```

//In the .h file:
class A2 : public CBase_A2 {
private: //My data members:
    int nt;
    unsigned char chr;
    float flt[7];
    int numDbl;

```

(continues on next page)

(continued from previous page)

```

    double *dbl;
public:
    //...other declarations

    virtual void pup(PUP::er &p);
};

//In the .C file:
void A2::pup(PUP::er &p)
{
    // The runtime will automatically call CBase_A2::pup()
    p|nt;
    p|chr;
    p|flt,7);
    p|numDbl;
    if (p.isUnpacking()) dbl=new double[numDbl];
    p(dbl,numDbl);
}

```

The default assumption, as used in the example above, for the object state at PUP time is that a chare, and its member objects, could be migrated at any time while it is inactive, i.e. not executing an entry method. Actual migration time can be controlled (see section 2.2.6) to be less frequent. If migration timing is fully user controlled, e.g., at the end of a synchronized load balancing step, then PUP implementation can be simplified to only transport “live” ephemeral data matching the object state which coincides with migration. More intricate state based PUPing, for objects whose memory footprint varies substantially with computation phase, can be handled by explicitly maintaining the object’s phase in a member variable and implementing phase conditional logic in the PUP method (see section 2.3.9).

Marshalling User Defined Data Types via PUP

Parameter marshalling requires serialization and is therefore implemented using the PUP framework. User defined data types passed as parameters must abide by the standard PUP contract (see section 2.2.5).

A simple example of using PUP to marshall user defined data types follows:

```

class Buffer {
public:
    //...other declarations
    void pup(PUP::er &p) {
        // remember to pup your superclass if there is one
        p|size;
        if (p.isUnpacking())
            data = new int[size];
        PUParray(p, data, size);
    }

private:
    int size;
    int *data;
};

// In some .ci file
entry void process(Buffer &buf);

```

For efficiency, arrays are always copied as blocks of bytes and passed via pointers. This means classes that need their pup routines to be called, such as those with dynamically allocated data or virtual methods cannot be passed as arrays-

use STL vectors to pass lists of complicated user-defined classes. For historical reasons, pointer-accessible structures cannot appear alone in the parameter list (because they are confused with messages).

The order of marshalling operations on the send side is:

- Call “p | a” on each marshalled parameter with a sizing PUP::er.
- Compute the lengths of each array.
- Call “p | a” on each marshalled parameter with a packing PUP::er.
- memcpy each arrays’ data.

The order of marshalling operations on the receive side is:

- Create an instance of each marshalled parameter using its default constructor.
- Call “p | a” on each marshalled parameter using an unpacking PUP::er.
- Compute pointers into the message for each array.

Finally, very large structures are most efficiently passed via messages, because messages are an efficient, low-level construct that minimizes copying and overhead; but very complicated structures are often most easily passed via marshalling, because marshalling uses the high-level pup framework.

See `examples/charm++/PUP/HeapPUP`

2.2.6 Load Balancing

Load balancing in Charm++ is enabled by its ability to place and migrate chares (note that only chare array elements are relocatable via migration, singleton chares can be created on a particular PE via seed balancing to balance load). Typical application usage to exploit this feature will construct many more chares than processors, and enable their runtime migration.

Iterative applications, which are commonplace in physical simulations, are the most suitable target for Charm++’s measurement based load balancing techniques. Such applications may contain a series of time-steps, and/or iterative solvers that run to convergence. For such computations, typically, the heuristic principle that we call “principle of persistence” holds: the computational loads and communication patterns between objects (chares) tend to persist over multiple iterations, even in dynamic applications. In such cases, the recent past is a good predictor of the near future. Measurement-based chare migration strategies are useful in this context. Currently these apply to chare-array elements, but they may be extended to chares in the future.

For applications without such iterative structure, or with iterative structure, but without predictability (i.e. where the principle of persistence does not apply), Charm++ supports “seed balancers” that move “seeds” for new chares among processors (possibly repeatedly) to achieve load balance. These strategies are currently available for both chares and chare-arrays. Seed balancers were the original load balancers provided in Charm since the late 80’s. They are extremely useful for state-space search applications, and are also useful in other computations, as well as in conjunction with migration strategies.

For iterative computations when there is a correlation between iterations/steps, but either it is not strong, or the machine environment is not predictable (due to noise from OS interrupts on small time steps, or time-shared desktop machines), one can use a combination of the two kinds of strategies. The baseline load balancing is provided by migration strategies, but in each iteration one also spawns off work in the form of chares that can run on any processor. The seed balancer will handle such work as it arises.

Examples are in `examples/charm++/load_balancing` and `tests/charm++/load_balancing`

Measurement-based Object Migration Strategies

In Charm++, chare array elements can migrate from processor to processor at runtime. Object migration can potentially improve the performance of the parallel program by migrating objects from overloaded processors to underloaded ones.

Charm++ implements a generic, measurement-based load balancing framework which automatically instruments all Charm++ objects, collecting computational load and communication structure during execution and storing them into a load balancing database (this only happens when a load balancer is activated during execution, see section 2.2.6). This instrumentation starts automatically at the beginning of application execution by default. It can be disabled at startup by passing the `+LBOff` flag at runtime, and toggled from the application by calling `LBTurnInstrumentOn()` and `LBTurnInstrumentOff()`, enabling or disabling instrumentation on the calling PE.

Charm++ then provides a collection of load balancing strategies whose job it is to decide on a new mapping of objects to processors based on the information from the database. Such measurement based strategies are efficient when we can reasonably assume that objects in a Charm++ application tend to exhibit temporal correlation in their computation and communication patterns, i.e. that the future can be to some extent predicted using the historical measurement data, allowing effective measurement-based load balancing without application-specific knowledge.

The instrumentation stored in the load balancing database is cleared immediately following each time load balancing completes. This means that each invocation of load balancing uses only data measured since the last invocation, providing adaptive and responsive results even for dynamic applications.

Two key terms in the Charm++ load balancing framework are:

- The **load balancing manager** provides the interface of almost all load balancing calls. On each processor, it manages the load balancing database, which stores the instrumented load data, and controls and invokes the selected load balancing strategies. It is implemented as a chare group called `LBManager`.
- A **load balancing strategy** gathers the relevant load data, runs a decision algorithm and produces the new mapping of the objects. Charm++ supports several kinds of strategies:
 - a. Configurable, hierarchical load balancers using `TreeLB`
 - b. Fully distributed load balancers
 - c. (*deprecated*) Centralized load balancers using `CentralLB`
 - d. (*deprecated*) Hierarchical load balancers using `HybridBaseLB`

Available Load Balancing Strategies

`TreeLB` and its pluggable strategies supersede the previous implementations of centralized and hierarchical load balancing. To use `TreeLB`, the user selects one of several trees. Each level corresponds to a different division of the overall execution (e.g. PE, process). Each level is configurable with a list of strategies, frequency, and other parameters. See 2.2.6 below for more detail, along with configuration and execution instructions. The following strategies can be used with `TreeLB` (the old runtime selection syntax still works and is specified in parentheses, it uses the new `TreeLB` versions with a two level `PE_Root` tree rooted at PE 0, emulating the centralized structure of the old implementation):

- **Greedy**: Uses a greedy algorithm that iterates over the objects and assigns the heaviest remaining object to the least loaded processor. (Old: `+balancer GreedyLB`)
- **GreedyRefine**: Uses a greedy algorithm that assigns the heaviest remaining object to the least loaded processor when it is currently assigned to a heavily loaded processor, otherwise leaves the object on its current processor to limit migrations. It takes an optional argument `tolerance` via the configuration file, which specifies the tolerance it should allow above the maximum load Greedy would produce (e.g. `1.1` allows the maximum load to be 10% higher than Greedy's max load). (Old: `+balancer GreedyRefineLB`)

- **RefineA, RefineB:** Moves objects away from the most overloaded processors to reach average, limits the number of objects migrated. RefineA allows a heavy object to go to any of the lightly loaded PEs, while RefineB always moves the heaviest remaining object to the lightest loaded PE. (Old: `+balancer RefineLB` runs RefineA)

Listed below are load balancers intended for diagnostic purposes:

- **Dummy:** Does nothing, does not move objects at all. (Old: `+balancer DummyLB`)
- **Random:** Randomly assigns objects to processors. (Old: `+balancer RandCentLB`)
- **Rotate:** Moves objects to the next available PE every time it is called. It is useful for debugging PUP routines and other migration related bugs. (Old: `+balancer RotateLB`)

The following centralized communication-aware load balancers do not yet use TreeLB, but continue to be available using the old CentralLB infrastructure:

- **RecBipartLB:** Uses recursive bipartitioning to partition the object communication graph. (`+balancer RecBipartLB`)
- **MetisLB:** Uses [METIS](#) to partition the object communication graph. METIS is distributed with Charm++, so there is no need to separately get this dependence. (`+balancer MetisLB`)
- **ScotchLB:** Uses the [SCOTCH](#) library for partitioning the object communication graph, while also taking object load imbalance into account. SCOTCH is not distributed with Charm++, so end users must download and build the library from the above link in order to use this load balancer. Because of this dependence, ScotchLB is not built by default; it can be built by running `make ScotchLB` in the Charm++ build folder (e.g. `netlrts-linux-x86_64-smp/`). If SCOTCH is installed in a non-standard location, use the `-incdir` and `-libdir` build time options to point to the include and library directories used, respectively. (`+balancer ScotchLB`)

In distributed approaches, the strategy executes across multiple PEs, providing scalable computational and communication performance.

Listed below are the distributed load balancers:

- **DistributedLB:** A load balancer which uses partial information about underloaded and overloaded processors in the system to do probabilistic transfer of load. This is a refinement based strategy. (`+balancer DistributedLB`)

Custom strategies should be built using TreeLB or DistBaseLB (the base class for DistributedLB). Custom strategies that are based on CentralLB or HybridBaseLB will continue to be supported for now, but support for these will likely be dropped in a future release.

All built-in load balancers that do not require external dependencies (that is, all of the above load balancers except for ScotchLB) are built by default. To use load balancing, users must link a load balancing module with their application and pass the appropriate runtime flags. See [2.2.6](#) for details on how to use load balancing.

Users can choose any load balancing strategy they think is appropriate for their application. We recommend using TreeLB with GreedyRefine for applications in general. For applications where the cost of migrating objects is very high, say, due to frequent load balancing to handle frequent load changes or due to size of data in the object being large, a strategy which favors migration minimization at the cost of balance (eg: RefineLB) is more suitable. DistributedLB is suitable for a large number of nodes. Communication-aware load balancers like MetisLB and RecBipartLB are suitable for communication intensive applications. The compiler and runtime options are described in section [2.2.6](#).

TreeLB and its Configuration

TreeLB allows for user-configurable hierarchical load balancing. While the legacy centralized strategies above are still supported, TreeLB allows load balancing to be performed at different levels and frequencies in a modular way. TreeLB includes several kinds of trees: the 2-level tree consists of PE and root levels (essentially the same as centralized load balancing), the 3-level tree consists of PE, process, and root levels, and the 4-level tree consists of PE, process, process

group, and root levels (process groups are collections of consecutive processes; the number of groups is configurable, see below). Each level only balances load within its corresponding domain; for example, for the 3-level PE-Process-Root tree: during process steps, each process runs the specified LB strategy over only the PEs and objects contained within the process, while, at root steps, the root strategy is run over all PEs and objects in the job. Supposing the root step frequency is 3, the root strategy is GreedyRefine, and the process strategy is Greedy, LB would proceed as follows:

LB Step	LB Action
0	Each process runs Greedy over its own PEs
1	Each process runs Greedy over its own PEs
2	Root PE runs GreedyRefine over all PEs
3	Each process runs Greedy over its own PEs
...	...

The load balancing strategy to be used at each level and frequency at which to invoke LB at each level can be specified using a JSON configuration file with name `treelb.json` or by specifying the JSON file name using command line option `+TreeLBFile`. We provide examples of some configuration files below:

Creating a 2-level tree that first uses the Greedy strategy and then the GreedyRefine strategy at the root:

```
{
  "tree": "PE_Root",
  "root": {
    "pe": 0,
    "strategies": ["Greedy", "GreedyRefine"]
  }
}
```

Creating a 3-level tree that uses the Greedy strategy at the process level and the GreedyRefine strategy at the root, which runs only every three steps:

```
{
  "tree": "PE_Process_Root",
  "root": {
    "pe": 0,
    "step_freq": 3,
    "strategies": ["GreedyRefine"]
  },
  "process": {
    "strategies": ["Greedy"]
  }
}
```

Creating a 4-level tree that uses the GreedyRefine strategy at the process and process group levels. The number of user-specified process groups is four in this example. A strategy is not allowed at root level for a 4-level tree since communicating all object load information to the root can be expensive given the size of the PE tree. Instead, a scheme where coarsened representations of the subtrees exchange load tokens is used at the root level. Load is balanced at the process group level every five steps and at the root level every ten steps. Note also that the process group usage of GreedyRefine provides a custom parameter to the strategy. This parameter will only be used for the process group level version of GreedyRefine, not the process level version.

```
{
  "tree": "PE_Process_ProcessGroup_Root",
```

(continues on next page)

(continued from previous page)

```

"root":
{
  "pe": 0,
  "step_freq": 10
},
"processgroup":
{
  "step_freq": 5,
  "strategies": ["GreedyRefine"],
  "num_groups": 4,
  "GreedyRefine":
  {
    "tolerance": 1.03
  }
},
"process":
{
  "strategies": ["GreedyRefine"]
}
}

```

TreeLB Configuration Parameters

The following parameters may be used to specify the configuration of TreeLB:

- **tree (required)**: String specifying the tree to use. Can be one of “PE_Root”, “PE_Process_Root”, or “PE_Process_ProcessGroup_Root”.
- **root (required)**: The configuration block for the root level of the tree.
 - **pe**: Integer specifying the root PE. (default = 0)
 - **token_passing**: Boolean specifying whether to use the coarsened token passing strategy or not, only allowed for the “PE_Process_ProcessGroup_Root” tree. If false, load will only be balanced within process groups at most, never across the whole job. (default = true)
- **processgroup (required for “PE_Process_ProcessGroup_Root” tree)**: The configuration block for the process group level of the tree.
 - **num_groups (required)**: Integer specifying the number of process groups to create.
- **process (required for “PE_Process_Root” and “PE_Process_ProcessGroup_Root” trees)**: The configuration block for the process level of the tree.
- **mcast_bfactor**: 8-bit integer specifying the branching factor of the communication tree used to send inter-subtree migrations for the 4-level tree’s token passing scheme. (default = 4)

The **root**, **processgroup**, and **process** blocks may include the following tree level configuration parameters:

- **strategies (required except for root level of “PE_Process_ProcessGroup_Root” tree)**: List of strings specifying which LB strategies to run at the given level.
- **repeat_strategies**: Boolean specifying if the whole list of strategies should be repeated or not. If true, start back at the beginning when the end of **strategies** is reached, otherwise keep repeating the last strategy (e.g. for **"strategies": ["1", "2"]**, true would result in 1, 2, 1, 2, ...; false in 1, 2, 2, 2, ...). (default = false)
- **step_freq**: Integer specifying frequency at which to balance at the given level of the tree. Not allowed to be specified on the level immediately above the PE level of the tree, which implicitly has a value of 1. This value must be a multiple of the value for the level below it. For example, for the 4-level tree, this can be specified for the process group and root levels, and the value for the root level must be a multiple of the process group value.

If these values are given as 2 and 4, respectively, then load balancing will be performed at the following levels in sequence: process, process group, process, root, process, and so on. (default = max(value of the level below, 1))

- Individual strategies can also be configured using parameters in this file. These should be placed in a block with a key exactly matching the name of the load balancer, and can be parsed from within the strategy's constructor.

- GreedyRefine:

- * `tolerance`: Float specifying the tolerance GreedyRefine should allow above the maximum load of Greedy, e.g. 1.1 allows the maximum load to be 10% higher than Greedy's max load. (default = 1)

Metabaler to automatically schedule load balancing

Metabaler can be invoked to automatically decide when to invoke the load balancer, given the load-balancing strategy. Metabaler uses a linear prediction model to set the load balancing period based on observed load imbalance.

The runtime option `+MetaLB` can be used to invoke this feature to automatically invoke the load balancing strategy based on the imbalance observed. This option needs to be specified alongside the `+balancer` option to specify the load balancing strategy to use. Metabaler relies on the `AtSync()` calls specified in Section 2.2.6 to collect load statistics.

`+MetaLBModelDir <path-to-model>` can be used to invoke the Metabaler feature to automatically decide which load balancing strategy to invoke. A model trained on a generic representative load imbalance benchmark can be found in `charm/src/ck-ldb/rf_model`. Metabaler makes a decision on which load balancing strategy to invoke out of a subset of strategies, namely GreedyLB, RefineLB, HybridLB, DistributedLB, MetisLB and ScotchLB. For using the model based prediction in Metabaler, Charm++ needs to be built with all the above load balancing strategies, including ScotchLB, which relies on the external graph partitioning library SCOTCH specified in Section 2.2.6.

Load Balancing Chare Arrays

The load balancing framework is well integrated with chare array implementation - when a chare array is created, it automatically registers its elements with the load balancing framework. The instrumentation of compute time (WALL/CPU time) and communication pattern is done automatically and APIs are provided for users to trigger the load balancing. To use the load balancer, you must make your array elements migratable (see migration section above) and choose a load balancing strategy (see the section 2.2.6 for a description of available load balancing strategies).

There are two different ways to use load balancing for chare arrays to meet different needs of the applications. These methods are different in how and when a load balancing phase starts. The two methods are: **AtSync mode** and **periodic load balancing mode**. In *AtSync mode*, the application invokes the load balancer explicitly at an appropriate location (generally at a pre-existing synchronization boundary) to trigger load balancing by inserting a function call (`AtSync()`) in the application source code. In *periodic load balancing mode*, a user specifies only how often load balancing is to occur, using the `+LBPeriod` runtime parameter or the `LBManager::SetLBPeriod(double period)` call to specify the time interval.

The detailed APIs of these two methods are described as follows:

1. **AtSync mode:** Using this method, load balancing is triggered only at certain points in the execution, when the application invokes `AtSync()`, which is essentially a non-blocking barrier. In order to use AtSync mode, one should set the variable `usesAtSync` to true in the constructors of chare array elements that are participating in the AtSync barrier. When an element is ready to start load balancing, it calls `AtSync()`⁶. When all local elements that have set `usesAtSync` to true call `AtSync()`, the load balancer is triggered. (Note that when the load balancer is triggered, it is triggered for all array elements, even those without `usesAtSync` set to true. If they are migratable, then they should have PUP routines suitable for anytime migration.) Once all local migrations (both in and out) are completed, the load balancer calls the virtual function `ResumeFromSync()` on each of the local array elements participating in the AtSync barrier. This function is usually overridden by the application to trigger the resumption of execution.

⁶ `AtSync()` is a member function of class `ArrayElement`

Note that `AtSync()` is not a blocking call. The object may be migrated during the time between `AtSync()` and `ResumeFromSync()`. One can choose to let objects continue working with incoming messages; however, keep in mind the object may suddenly show up in another processor, so make sure no operations that could possibly prevent migration be performed. This is the automatic way of doing load balancing where the application does not need to define `ResumeFromSync()`.

The more commonly used approach is to force the object to be idle until load balancing finishes. The user calls `AtSync()` at the end of some iteration, then, when all participating elements reach that call, load balancing is triggered. The objects can start executing again when `ResumeFromSync()` is called. In this case, the user redefines `ResumeFromSync()` to trigger the next iteration of the application. This pattern effectively results in a barrier at load balancing time (see example here [2.2.6](#)).

Note: In `AtSync` mode, Applications that use dynamic insertion or deletion of array elements must not be doing so when any element calls `AtSync()`. This is because `AtSync` mode requires an application to have a fixed, known number of objects when determining if `AtSync()` has been called by all relevant objects in order to prevent race conditions (the implementation is designed to be robust against these issues and will often be able to handle them, but we make no guarantees if these rules are not obeyed). If using dynamic insertion, please ensure that insertions and calls to `AtSync()` cannot be interleaved and that `doneInserting()` is called after insertions are complete and before any element calls `AtSync()`. Insertions and/or deletions may begin again after load balancing is complete (i.e. `ResumeFromSync()` is called for an object on the given PE for insertions or for the object in question for deletions).

2. **Periodic load balancing mode:** This mode uses a timer to perform load balancing periodically at a user-specified interval. In order to use this mode, the user must either provide the `+LBPeriod {period}` runtime option or set the period from the application using `LBManager::SetLBPeriod(double period)` on every PE, the *period* argument specifying the minimum time between consecutive LB invocations in seconds in both cases. For example, `+LBPeriod 10.5` can be used to invoke load balancing roughly every 10.5 seconds. Additionally, no array element can have `usesAtSync` set to true. In this mode, array elements may be asked to migrate at any time, provided that they are not in the middle of executing an entry method. Thus, the PUP routines for array elements must migrate all data needed to reconstruct the object at any point in its lifecycle (as opposed to `AtSync` mode, where PUP routines for load balancing migration are only called after `AtSync()` and before `ResumeFromSync()`, so they can make some assumptions about state).

Note: Dynamic insertion works with periodic load balancing with no issues. However, dynamic deletion does not, since deletion may occur while the load balancing strategy is running.

Migrating objects

Load balancers migrate objects automatically. For an array element to migrate, user can refer to [Section 2.2.5](#) for how to write a “pup” for an array element.

In general one needs to pack the whole snapshot of the member data in an array element in the pup subroutine. This is because the migration of the object may happen at any time. In certain load balancing schemes where the user explicitly controls when load balancing occurs, the user may choose to pack only a part of the data and may skip temporary data.

An array element can migrate by calling the `migrateMe(destination processor)` member function- this call must be the last action in an element entry method. The system can also migrate array elements for load balancing (see the section [2.2.6](#)).

To migrate your array element to another processor, the Charm++ runtime will:

- Call your `ckAboutToMigrate` method

- Call your pup method with a sizing PUP::er to determine how big a message it needs to hold your element.
- Call your pup method again with a packing PUP::er to pack your element into a message.
- Call your element's destructor (deleting the old copy).
- Send the message (containing your element) across the network.
- Call your element's migration constructor on the new processor.
- Call your pup method on with an unpacking PUP::er to unpack the element.
- Call your ckJustMigrated method

Migration constructors, then, are normally empty- all the unpacking and allocation of the data items is done in the element's pup routine. Deallocation is done in the element destructor as usual.

Other utility functions

There are several utility functions that can be called in applications to configure the load balancer, etc. These functions are:

- **LBTurnInstrumentOn()** and **LBTurnInstrumentOff()**: are plain C functions to control the load balancing statistics instrumentation on or off on the calling processor. No implicit broadcast or synchronization exists in these functions. Fortran interface: **FLBTURNINSTRUMENTON()** and **FLBTURNINSTRUMENTOFF()**.
- **setMigratable(bool migratable)**: is a member function of array element. This function can be called in an array element constructor to tell the load balancer whether this object is migratable or not⁷.
- **double LBManager::GetLBPeriod()**: returns the current load balancing period when using *periodic load balancing mode*. Returns -1.0 when no period is set or when using a different LB mode.
- **LBManager::SetLBPeriod(double s)**: The **SetLBPeriod** function can be called anywhere (even in Charm++ initnodes or initprocs) to change the load balancing period time when using *periodic load balancing mode*. It tells the load balancer to use the given period *s* as the new minimum time between load balancing invocations. It may take up to one full cycle of load balancing before the new period comes into effect (so up to the the period after the next load balancing invocation completes). This call should be made at least once on every PE when setting the period. If no elements have *usesAtSync* set to true and no LB period was set on the command line, this call will also enable *periodic load balancing mode*. Here is how to use it:

```
// if used in an array element:
LBManager* lbmgr = getLBMgr();
lbmgr->SetLBPeriod(5.0);

// If used outside an array element, since it's a static member function:
LBManager::SetLBPeriod(5.0);
```

Compiler and runtime options to use load balancing

Load balancing strategies are implemented as libraries in Charm++. This allows programmers to easily experiment with different existing strategies by simply linking a pool of strategy modules and choosing one to use at runtime via a command line option.

Note: linking a load balancing module is distinct from activating it:

- link an LB module: to link an load balancing module (library) at compile time. You can link against multiple LB libraries.

⁷ Currently not all load balancers recognize this setting though.

- activate an LB: to actually ask the runtime to create an LB strategy and use it for a given run. You can only activate load balancers that have been already been linked in at compile time. The special `-balancer {balancer name}` link time argument both links a module and activates it at runtime by default.

Below are the descriptions about the compiler and runtime options:

1. Compile time options: (to `charmcc`)

- `-module TreeLB -module RecBipartLB ...`
links the listed LB modules into an application, which can then be used at runtime via the `+balancer` option.
- `-module CommonLBs`
links a special module `CommonLBs` which includes some commonly used Charm++ built-in load balancers. This set includes the following commonly used load balancers: *TreeLB* (usable with *Greedy*, *GreedyRefine*, *RefineA*, *RefineB*, *Dummy*, *Random*, and *Rotate*) and *DistributedLB*.
- `-module EveryLB`
links a special module `EveryLB` which includes all Charm++ load balancers built by default. This set includes everything specified in `CommonLBs` plus *MetisLB* and *RecBipartLB*.
- `-balancer MetisLB`
links the given load balancer *and activates* it for use at runtime. This is equivalent to using `-module MetisLB` at compile time and then `+balancer MetisLB` at runtime.
- `-balancer GreedyLB -balancer RefineLB`
links both listed balancers, then invokes *GreedyLB* at the first load balancing step and *RefineLB* in all subsequent load balancing steps.

The list of existing load balancers is given in Section 2.2.6. Note: you can have multiple `-module *LB` options. LB modules are linked into a program, but they are not activated automatically at runtime. Using `-balancer A` at compile time will activate load balancer A automatically at runtime. Having `-balancer A` implies `-module A`, so you don't have to write `-module A` again, although that is not invalid. Using `CommonLBs` is a convenient way to link against the commonly used existing load balancers.

2. Runtime options:

Runtime balancer selection options are similar to the compile time options as described above, but they can be used to override those compile time options.

- `+balancer help`
displays all available balancers that have been linked in.
- `+balancer DistributedLB`
invokes *DistributedLB*
- `+balancer GreedyLB +balancer RefineLB`
invokes *GreedyLB* at the first load balancing step and *RefineLB* in all subsequent load balancing steps.

Note: The `+balancer` option works only if you have already linked the corresponding load balancer module at compile time. Providing `+balancer` with an invalid LB name will result in a runtime error. When you have used `-balancer A` rather than `-module A` as a compile time option, you do not need to use `+balancer A` again to activate it at runtime. However, you can use `+balancer B` to override the compile time option and choose to activate B instead of A (assuming that B was also linked into the application).

3. Other useful runtime options

There are a few other runtime options for load balancing that may be useful:

- `+LBPeriod {seconds}`

When not using AtSync mode (meaning no chare array element in the application has `usesAtSync` set to true), this option enables *periodic load balancing mode*. The argument {seconds} specifies the interval for invoking load balancing in seconds; it can be any floating point number. Note that this sets the *minimum period* between two consecutive load balancing steps.

- **+LBDebug {verbose level}**
{verbose level} can be any non-negative integer. 0 is equivalent to not passing this flag at all. When active, the load balancer will output load balancing information to stdout when it runs. The bigger {verbose level} is, the more verbose the output (the number of levels varies by load balancer).
- **+LBSameCpus**
This option simply tells load balancer that all processors are of same speed. The load balancer will then skip the measurement of CPU speed at runtime. This is the default.
- **+LBTestPESpeed**
This option tells the load balancer to test the speed of all processors at runtime. The load balancer may use this measurement to perform speed-aware load balancing.
- **+LBObjOnly**
This tells load balancer to ignore processor background load when making migration decisions.
- **+LBSyncResume**
After load balancing step, normally a processor can resume computation once all objects are received on that processor, even when other processors are still working on migrations. If this turns out to be a problem, that is when some processors start working on computation while the other processors are still busy migrating objects, then this option can be used to force a global barrier on all processors to make sure that processors can only resume computation after migrations are completed on all processors.
- **+LBOff**
This option turns off load balancing instrumentation of both CPU and communication usage at startup time.
- **+LBCommOff**
This option turns off load balancing instrumentation of communication at startup time. The instrument of CPU usage is left on.

Seed load balancers - load balancing Chares at creation time

Seed load balancing involves the movement of object creation messages, or “seeds”, to create a balance of work across a set of processors. This seed load balancing scheme is used to balance chares at creation time. After the chare constructor is executed on a processor, the seed balancer does not migrate it. Depending on the movement strategy, several seed load balancers are available now. Examples can be found `examples/charm++/NQueen`.

1. *random*
A strategy that places seeds randomly when they are created and does no movement of seeds thereafter. This is used as the default seed load balancer.
2. *neighbor*
A strategy which imposes a virtual topology on the processors, load exchange happens among neighbors only. The overloaded processors initiate the load balancing and send work to its neighbors when it becomes overloaded. The default topology is mesh2D, one can use command line option to choose other topology such as ring, mesh3D and dense graph.
3. *spray*
A strategy which imposes a spanning tree organization on the processors, results in communication via global reduction among all processors to compute global average load via periodic reduction. It uses averaging of loads to determine how seeds should be distributed.

4. *workstealing*

A strategy that the idle processor requests a random processor and steal chares.

Other strategies can also be explored by following the simple API of the seed load balancer.

Compile and run time options for seed load balancers

To choose a seed load balancer other than the default *rand* strategy, use link time command line option **-balance foo**.

When using neighbor seed load balancer, one can also specify the virtual topology at runtime. Use **+LBTopo topo**, where *topo* can be one of: (a) ring, (b) mesh2d, (c) mesh3d and (d) graph.

To write a seed load balancer, name your file as *cldb.foo.c*, where *foo* is the strategy name. Compile it in the form of library under charm/lib, named as *libcldb-foo.a*, where *foo* is the strategy name used above. Now one can use **-balance foo** as compile time option to **charm** to link with the *foo* seed load balancer.

Simple Load Balancer Usage Example - Automatic with Sync LB

A simple example of how to use a load balancer in sync mode in one's application is presented below.

```

/** lbexample.ci */
mainmodule lbexample {
  readonly CProxy_Main mainProxy;
  readonly int nElements;

  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void done(void);
  };

  array [1D] LBExample {
    entry LBExample(void);
    entry void doWork();
  };
};

```

```

/** lbexample.C */
#include <stdio.h>
#include "lbexample.decl.h"

/*readonly*/ CProxy_Main mainProxy;
/*readonly*/ int nElements;

#define MAX_WORK_CNT 50
#define LB_INTERVAL 5

/*mainchare*/
class Main : public CBase_Main
{
private:
  int count;
public:
  Main(CkArgMsg* m)
  {
    /*....Initialization....*/
    mainProxy = thisProxy;

```

(continues on next page)

(continued from previous page)

```

    CProxy_LBExample arr = CProxy_LBExample::ckNew(nElements);
    arr.doWork();
};

void done(void)
{
    count++;
    if(count==nElements){
        CkPrintf("All done");
        CkExit();
    }
};

/*array [1D]*/
class LBExample : public CBase_LBExample
{
private:
    int workcnt;
public:
    LBExample()
    {
        workcnt=0;
        /* May initialize some variables to be used in doWork */
        //Must be set to true to make AtSync work
        usesAtSync = true;
    }

    LBExample(CkMigrateMessage *m) { /* Migration constructor -- invoked when chare_
↪migrates */ }

    /* Must be written for migration to succeed */
    void pup(PUP::er &p){
        p|workcnt;
        /* There may be some more variables used in doWork */
    }

    void doWork()
    {
        /* Do work proportional to the chare index to see the effects of LB */

        workcnt++;
        if(workcnt==MAX_WORK_CNT)
            mainProxy.done();

        if(workcnt%LB_INTERVAL==0)
            AtSync();
        else
            doWork();
    }

    void ResumeFromSync(){
        doWork();
    }
};

#include "lbexample.def.h"

```


2.2.7 Processor-Aware Chare Collections

So far, we have discussed chares separately from the underlying hardware resources to which they are mapped. However, for writing lower-level libraries or optimizing application performance it is sometimes useful to create chare collections where a single chare is mapped per specified resource used in the run. The `group`⁸ and `node group` constructs provide this facility by creating a collection of chares with a single chare (or branch) on each PE (in the case of groups) or process (for node groups).

Group Objects

Groups have a definition syntax similar to normal chares, and they have to inherit from the system-defined class `CBase_ClassName`, where `ClassName` is the name of the group's C++ class⁹.

Group Definition

In the interface (`.ci`) file, we declare

```
group Foo {
    // Interface specifications as for normal chares

    // For instance, the constructor ...
    entry Foo(parameters1);

    // ... and an entry method
    entry void someEntryMethod(parameters2);
};
```

The definition of the `Foo` class is given in the `.h` file, as follows:

```
class Foo : public CBase_Foo {
    // Data and member functions as in C++
    // Entry functions as for normal chares

public:
    Foo(parameters1);
    void someEntryMethod(parameters2);
};
```

Group Creation

Groups are created using `ckNew` like chares and chare arrays. Given the declarations and definitions of group `Foo` from above, we can create a group in the following manner:

```
CProxy_Foo fooProxy = CProxy_Foo::ckNew(parameters1);
```

One can also use `ckNew` to get a `CkGroupID` as shown below:

```
CkGroupID fooGroupID = CProxy_Foo::ckNew(parameters1);
```

A `CkGroupID` is useful to specify dependence in group creations using `CkEntryOptions`. For example, in the following code, the creation of group `GroupB` on each PE depends on the creation of `GroupA` on that PE.

⁸ Originally called *Branch Office Chare* or *Branched Chare*

⁹ Older, deprecated syntax allows groups to inherit directly from the system-defined class `Group`

```
// Create GroupA
CkGroupID groupAID = CProxy_GroupA::ckNew(parameters1);

// Create GroupB. However, for each PE, do this only
// after GroupA has been created on it

// Specify the dependency through a `CkEntryOptions' object
CkEntryOptions opts;
opts.setGroupDepID(groupAID);

// The last argument to `ckNew' is the `CkEntryOptions' object from above
CkGroupID groupBID = CProxy_GroupB::ckNew(parameters2, opts);
```

Note that there can be several instances of each group type. In such a case, each instance has a unique group identifier, and its own set of branches.

Method Invocation on Groups

An asynchronous entry method can be invoked on a particular branch of a group through a proxy of that group. If we have a group with a proxy `fooProxy` and we wish to invoke entry method `someEntryMethod` on that branch of the group which resides on PE `somePE`, we would accomplish this with the following syntax:

```
fooProxy[somePE].someEntryMethod(parameters);
```

This call is asynchronous and non-blocking; it returns immediately after sending the message. A message may be broadcast to all branches of a group (i.e., to all PEs) using the notation :

```
fooProxy.anotherEntryMethod(parameters);
```

This invokes entry method `anotherEntryMethod` with the given parameters on all branches of the group. This call is also asynchronous and non-blocking, and it, too, returns immediately after sending the message.

Finally, when running in SMP mode with multiple PEs per node, one can broadcast a message to the branches of a group local to the sending node:

```
CkWithinNodeBroadcast(CkIndex_Foo::bar(), msg, fooProxy);
```

Where `CkIndex_Foo::bar()` is the index of the entry method you wish to invoke, `msg` is the Charm++ message (section 2.3.1) you wish to send, and `fooProxy` is the proxy to the group you wish to send. As before, it is an asynchronous call which returns immediately. Furthermore, if the receiving entry method is marked as `[nokeep]` (2.3.1), the message pointer will be shared with each group chare instead of creating an independent copy per receiver.

Recall that each PE hosts a branch of every instantiated group. Sequential objects, chares and other groups can gain access to this *PE-local* branch using `ckLocalBranch()`:

```
GroupType *g=groupProxy.ckLocalBranch();
```

This call returns a regular C++ pointer to the actual object (not a proxy) referred to by the proxy `groupProxy`. Once a proxy to the local branch of a group is obtained, that branch can be accessed as a regular C++ object. Its public methods can return values, and its public data is readily accessible.

Let us end with an example use-case for groups. Suppose that we have a task-parallel program in which we dynamically spawn new chares. Furthermore, assume that each one of these chares has some data to send to the mainchare. Instead of creating a separate message for each chare's data, we create a group. When a particular chare finishes its work, it reports its findings to the local branch of the group. When all the chares on a PE have finished their work, the

local branch can send a single message to the main chare. This reduces the number of messages sent to the mainchare from the number of chares created to the number of processors.

For a more concrete example on how to use groups, please refer to `examples/charm++/histogram_group`. It presents a parallel histogramming operation in which chare array elements funnel their bin counts through a group, instead of contributing directly to a reduction across all chares.

NodeGroup Objects

The *node group* construct is similar to the group construct discussed above. Rather than having one chare per PE, a node group is a collection of chares with one chare per *process*, or *logical node*. Therefore, each logical node hosts a single branch of the node group. As with groups, node groups can be addressed via globally unique identifiers. Nonetheless, there are significant differences in the semantics of node groups as compared to groups and chare arrays. When an entry method of a node group is executed on one of its branches, it executes on *some* PE within the logical node. Also, multiple entry method calls can execute concurrently on a single node group branch. This makes node groups significantly different from groups and requires some care when using them.

NodeGroup Declaration

Node groups are defined in a similar way to groups.¹⁰ For example, in the interface file, we declare:

```
nodegroup NodeGroupType {
    // Interface specifications as for normal chares
};
```

In the `.h` file, we define `NodeGroupType` as follows:

```
class NodeGroupType : public CBase_NodeGroupType {
    // Data and member functions as in C++
    // Entry functions as for normal chares
};
```

Like groups, NodeGroups are identified by a globally unique identifier of type `CkGroupID`. Just as with groups, this identifier is common to all branches of the NodeGroup, and can be obtained from the inherited data member `thisgroup`. There can be many instances corresponding to a single NodeGroup type, and each instance has a different identifier, and its own set of branches.

Method Invocation on NodeGroups

As with chares, chare arrays and groups, entry methods are invoked on NodeGroup branches via proxy objects. An entry method may be invoked on a *particular* branch of a nodegroup by specifying a *logical node number* argument to the square bracket operator of the proxy object. A broadcast is expressed by omitting the square bracket notation. For completeness, example syntax for these two cases is shown below:

```
// Invoke 'someEntryMethod' on the i-th logical node of
// a NodeGroup whose proxy is 'myNodeGroupProxy':
myNodeGroupProxy[i].someEntryMethod(parameters);

// Invoke 'someEntryMethod' on all logical nodes of
// a NodeGroup whose proxy is 'myNodeGroupProxy':
myNodeGroupProxy.someEntryMethod(parameters);
```

¹⁰ As with groups, older syntax allows node groups to inherit from `NodeGroup` instead of a specific, generated “`CBase_`” class.

It is worth restating that when an entry method is invoked on a particular branch of a nodegroup, it may be executed by *any* PE in that logical node. Thus two invocations of a single entry method on a particular branch of a NodeGroup may be executed *concurrently* by two different PEs in the logical node. If this could cause data races in your program, please consult Section 2.2.7 (below).

NodeGroups and exclusive Entry Methods

Node groups may have exclusive entry methods. The execution of an exclusive entry method invocation is *mutually exclusive* with those of all other exclusive entry methods invocations. That is, an exclusive entry method invocation is not executed on a logical node as long as another exclusive entry method is executing on it. More explicitly, if a method M of a nodegroup NG is marked exclusive, it means that while an instance of M is being executed by a PE within a logical node, no other PE within that logical node will execute any other *exclusive* methods. However, PEs in the logical node may still execute *non-exclusive* entry method invocations. An entry method can be marked exclusive by tagging it with the exclusive attribute, as explained in Section 2.3.1.

Accessing the Local Branch of a NodeGroup

The local branch of a NodeGroup NG, and hence its member fields and methods, can be accessed through the method NG* CProxy_NG::ckLocalBranch() of its proxy. Note that accessing data members of a NodeGroup branch in this manner is *not* thread-safe by default, although you may implement your own mutual exclusion schemes to ensure safety. One way to ensure safety is to use node-level locks, which are described in the Converse manual.

NodeGroups can be used in a similar way to groups so as to implement lower-level optimizations such as data sharing and message reduction.

2.2.8 Initializations at Program Startup

initnode and initproc Routines

Some registration routines need be executed exactly once before the computation begins. You may choose to declare a regular C++ subroutine initnode in the .ci file to ask Charm++ to execute the routine exactly once on *every logical node* before the computation begins, or to declare a regular C++ subroutine initproc to be executed exactly once on *every PE*.

```
module foo {
  initnode void fooNodeInit(void);
  initproc void fooProcInit(void);
  chare bar {
    ...
    initnode void barNodeInit(void);
    initproc void barProcInit(void);
  };
};
```

This code will execute the routines fooNodeInit and static bar::barNodeInit once on every logical node and fooProcInit and bar::barProcInit on every PE before the main computation starts. Initnode calls are always executed before initproc calls. Both init calls (declared as static member functions) can be used in chares, chare arrays and groups.

Note that these routines should only implement registration and startup functionality, and not parallel computation, since the Charm++ run time system will not have started up fully when they are invoked. In order to begin the parallel computation, you should use a mainchare instead, which gets executed on only PE 0.

Event Sequence During Charm++ Startup

At startup, every Charm++ program performs the following actions, in sequence:

1. **Module Registration:** all modules given in the .ci files are registered in the order of their specification in the linking stage of program compilation. For example, if you specified “-module A -module B” while linking your Charm++ program, then module A is registered before module B at runtime.
2. **initnode, initproc Calls:** all initnode and initproc functions are invoked before the creation of Charm++ data structures, and before the invocation of any mainchares’ main() methods.
3. **readonly Variables:** readonly variables are initialized on PE 0 in the mainchare, following program order as given in the main() method. After initialization, they are broadcast to all other PEs making them available in the constructors groups, chares, chare arrays, etc. (see below.)
4. **Group and NodeGroup Creation:** on PE 0, constructors of these objects are invoked in program order. However, on all other PEs, their creation is triggered by messages. Since message order is not guaranteed in Charm++ program, constructors of groups and nodegroups should **not** depend on other Group or NodeGroup objects on a PE. However, if your program structure requires it, you can explicitly specify that the creation of certain Groups/NodeGroups depends on the creation of others, as described in Section 2.2.7. In addition, since those objects are initialized after the initialization of readonly variables, readonly variables can be used in the constructors of Groups and NodeGroups.
5. **Charm++ Array Creation:** the order in which array constructors are called on PEs is similar to that described for groups and nodegroups, above. Therefore, the creation of one array should **not** depend on other arrays. As Array objects are initialized last, their constructors can use readonly variables and local branches of Group or NodeGroup objects.

2.3 Advanced Programming Techniques

2.3.1 Optimizing Entry Method Invocation

Messages

Although Charm++ supports automated parameter marshalling for entry methods, you can also manually handle the process of packing and unpacking parameters by using messages. A message encapsulates all the parameters sent to an entry method. Since the parameters are already encapsulated, sending messages is often more efficient than parameter marshalling, and can help to avoid unnecessary copying. Moreover, assume that the receiver is unable to process the contents of the message at the time that it receives it. For example, consider a tiled matrix multiplication program, wherein each chare receives an A -tile and a B -tile before computing a partial result for $C = A \times B$. If we were using parameter marshalled entry methods, a chare would have to copy the first tile it received, in order to save it for when it has both the tiles it needs. Then, upon receiving the second tile, the chare would use the second tile and the first (saved) tile to compute a partial result. However, using messages, we would just save a *pointer* to the message encapsulating the tile received first, instead of the tile data itself.

Managing the memory buffer associated with a message. As suggested in the example above, the biggest difference between marshalled parameters and messages is that an entry method invocation is assumed to *keep* the message that it is passed. That is, the Charm++ runtime system assumes that code in the body of the invoked entry method will explicitly manage the memory associated with the message that it is passed. Therefore, in order to avoid leaking memory, the body of an entry method must either delete the message that it receives, or save a pointer to it, and delete it a later point in the execution of the code.

Moreover, in the Charm++ execution model, once you pass a message buffer to the runtime system (via an asynchronous entry method invocation), you should *not* reuse the buffer. That is, after you have passed a message buffer into an asynchronous entry method invocation, you shouldn’t access its fields, or pass that same buffer into a second

entry method invocation. Note that this rule doesn't preclude the *single reuse* of a received message - consider being inside the body of an entry method invocation i_1 , which has received the message buffer m_1 as an input parameter. Then, m_1 may be passed to an asynchronous entry method invocation i_2 . However, once i_2 has been issued with m_1 as its input parameter, m_1 cannot be used in any further entry method invocations.

Several kinds of message are available. Regular Charm++ messages are objects of *fixed size*. One can have messages that contain pointers or variable length arrays (arrays with sizes specified at runtime) and still have these pointers as valid when messages are sent across processors, with some additional coding. Also available is a mechanism for assigning *priorities* to a message regardless of its type. A detailed discussion of priorities appears later in this section.

Message Types

Fixed-Size Messages. The simplest type of message is a *fixed-size* message. The size of each data member of such a message should be known at compile time. Therefore, such a message may encapsulate primitive data types, user-defined data types that *don't* maintain pointers to memory locations, and *static* arrays of the aforementioned types.

Variable-Size Messages. Very often, the size of the data contained in a message is not known until runtime. For such scenarios, you can use variable-size (*varsize*) messages. A *varsize* message can encapsulate several arrays, each of whose size is determined at run time. The space required for these encapsulated, variable length arrays is allocated with the entire message comprises a contiguous buffer of memory.

Packed Messages. A *packed* message is used to communicate non-linear data structures via messages. However, we defer a more detailed description of their use to Section 2.3.1.

Using Messages In Your Program

There are five steps to incorporating a (fixed or varsize) message type in your Charm++ program: (1) Declare message type in .ci file; (2) Define message type in .h file; (3) Allocate message; (4) Pass message to asynchronous entry method invocation and (5) Deallocate message to free associated memory resources.

Declaring Your Message Type. Like all other entities involved in asynchronous entry method invocation, messages must be declared in the .ci file. This allows the Charm++ translator to generate support code for messages. Message declaration is straightforward for fixed-size messages. Given a message of type `MyFixedSizeMsg`, simply include the following in the .ci file:

```
message MyFixedSizeMsg;
```

For varsize messages, the .ci declaration must also include the names and types of the variable-length arrays that the message will encapsulate. The following example illustrates this requirement. In it, a message of type `MyVarsizeMsg`, which encapsulates three variable-length arrays of different types, is declared:

```
message MyVarsizeMsg {  
    int arr1[];  
    double arr2[];  
    MyPointerlessStruct arr3[];  
};
```

Defining Your Message Type. Once a message type has been declared to the Charm++ translator, its type definition must be provided. Your message type must inherit from a specific generated base class. If the type of your message is `T`, then `class T` must inherit from `CMessage_T`. This is true for both fixed and varsize messages. As an example, for our fixed size message type `MyFixedSizeMsg` above, we might write the following in the .h file:

```
class MyFixedSizeMsg : public CMessage_MyFixedSizeMsg {  
    int var1;
```

(continues on next page)

(continued from previous page)

```
MyPointerlessStruct var2;
double arr3[10];

// Normal C++ methods, constructors, etc. go here
};
```

In particular, note the inclusion of the static array of doubles, `arr3`, whose size is known at compile time to be that of ten doubles. Similarly, for our example varsize message of type `MyVarsizeMsg`, we would write something like:

```
class MyVarsizeMsg : public CMessage_MyVarsizeMsg {
// variable-length arrays
int *arr1;
double *arr2;
MyPointerlessStruct *arr3;

// members that are not variable-length arrays
int x,y;
double z;

// Normal C++ methods, constructors, etc. go here
};
```

Note that the `.h` definition of the class type must contain data members whose names and types match those specified in the `.ci` declaration. In addition, if any of data members are private or protected, it should declare class `CMessage_MyVarsizeMsg` to be a friend class. Finally, there are no limitations on the member methods of message classes, except that the message class may not redefine operators `new` or `delete`.

Thus the `mtype` class declaration should be similar to:

Creating a Message. With the `.ci` declaration and `.h` definition in place, messages can be allocated and used in the program. Messages are allocated using the C++ `new` operator:

```
MessageType *msgptr =
new [(int) sz1, (int) sz2, ... , (int) priobits=0] MessageType[(constructor arguments)];
```

The arguments enclosed within the square brackets are optional, and are used only when allocating messages with variable length arrays or prioritized messages. These arguments are not specified for fixed size messages. For instance, to allocate a message of our example message `MyFixedSizeMsg`, we write:

```
MyFixedSizeMsg *msg = new MyFixedSizeMsg(<constructor args>);
```

In order to allocate a varsize message, we must pass appropriate values to the arguments of the overloaded `new` operator presented previously. Arguments `sz1`, `sz2`, ... denote the size (in number of elements) of the memory blocks that need to be allocated and assigned to the pointers (variable-length arrays) that the message contains. The `priobits` argument denotes the size of a bitvector (number of bits) that will be used to store the message priority. So, if we wanted to create `MyVarsizeMsg` whose `arr1`, `arr2` and `arr3` arrays contain 10, 20 and 7 elements of their respective types, we would write:

```
MyVarsizeMsg *msg = new (10, 20, 7) MyVarsizeMsg(<constructor args>);
```

Further, to add a 32-bit priority bitvector to this message, we would write:

```
MyVarsizeMsg *msg = new (10, 20, 7, sizeof(uint32_t)*8) VarsizeMessage;
```

Notice the last argument to the overloaded `new` operator, which specifies the number of bits used to store message priority. The section on prioritized execution (Section 2.3.1) describes how priorities can be employed in your program.

Another version of the overloaded new operator allows you to pass in an array containing the size of each variable-length array, rather than specifying individual sizes as separate arguments. For example, we could create a message of type `MyVarsizeMsg` in the following manner:

```
int sizes[3];
sizes[0] = 10;           // arr1 will have 10 elements
sizes[1] = 20;           // arr2 will have 20 elements
sizes[2] = 7;            // arr3 will have 7 elements

MyVarsizeMsg *msg = new(sizes, 0) MyVarsizeMsg(<constructor args>); // 0 priority bits
```

Sending a Message. Once we have a properly allocated message, we can set the various elements of the encapsulated arrays in the following manner:

```
msg->arr1[13] = 1;
msg->arr2[5] = 32.82;
msg->arr3[2] = MyPointerlessStruct();
// etc.
```

And pass it to an asynchronous entry method invocation, thereby sending it to the corresponding chare:

```
myChareArray[someIndex].foo(msg);
```

When a message is *sent*, i.e. passed to an asynchronous entry method invocation, the programmer relinquishes control of it; the space allocated for the message is freed by the runtime system. However, when a message is *received* at an entry point, it is *not* freed by the runtime system. As mentioned at the start of this section, received messages may be reused or deleted by the programmer. Finally, messages are deleted using the standard C++ delete operator.

Message Packing

The Charm++ interface translator generates implementation for three static methods for the message class `CMessage_mtype`. These methods have the prototypes:

```
static void* alloc(int msgnum, size_t size, int* array, int priobits);
static void* pack(mtype*);
static mtype* unpack(void*);
```

One may choose not to use the translator-generated methods and may override these implementations with their own `alloc`, `pack` and `unpack` static methods of the `mtype` class. The `alloc` method will be called when the message is allocated using the C++ new operator. The programmer never needs to explicitly call it. Note that all elements of the message are allocated when the message is created with `new`. There is no need to call `new` to allocate any of the fields of the message. This differs from a packed message where each field requires individual allocation. The `alloc` method should actually allocate the message using `CkAllocMsg`, whose signature is given below:

```
void *CkAllocMsg(int msgnum, int size, int priobits);
```

For varsize messages, these static methods `alloc`, `pack`, and `unpack` are generated by the interface translator. For example, these methods for the `VarsizeMessage` class above would be similar to:

```
// allocate memory for varmessage so charm can keep track of memory
static void* alloc(int msgnum, size_t size, int* array, int priobits)
{
    int totalsize, first_start, second_start;
    // array is passed in when the message is allocated using new (see below).
    // size is the amount of space needed for the part of the message known
```

(continues on next page)

(continued from previous page)

```

// about at compile time. Depending on their values, sometimes a segfault
// will occur if memory addressing is not on 8-byte boundary, so altered
// with ALIGN8
first_start = ALIGN8(size); // 8-byte align with this macro
second_start = ALIGN8(first_start + array[0]*sizeof(int));
totalsize = second_start + array[1]*sizeof(double);
VarsizeMessage* newMsg =
    (VarsizeMessage*) CkAllocMsg(msgnum, totalsize, priobits);
// make firstArray point to end of newMsg in memory
newMsg->firstArray = (int*) ((char*)newMsg + first_start);
// make secondArray point to after end of firstArray in memory
newMsg->secondArray = (double*) ((char*)newMsg + second_start);

return (void*) newMsg;
}

// returns pointer to memory containing packed message
static void* pack(VarsizeMessage* in)
{
    // set firstArray an offset from the start of in
    in->firstArray = (int*) ((char*)in->firstArray - (char*)in);
    // set secondArray to the appropriate offset
    in->secondArray = (double*) ((char*)in->secondArray - (char*)in);
    return in;
}

// returns new message from raw memory
static VarsizeMessage* VarsizeMessage::unpack(void* inbuf)
{
    VarsizeMessage* me = (VarsizeMessage*)inbuf;
    // return first array to absolute address in memory
    me->firstArray = (int*) ((size_t)me->firstArray + (char*)me);
    // likewise for secondArray
    me->secondArray = (double*) ((size_t)me->secondArray + (char*)me);
    return me;
}

```

The pointers in a varsize message can exist in two states. At creation, they are valid C++ pointers to the start of the arrays. After packing, they become offsets from the address of the pointer variable to the start of the pointed-to data. Unpacking restores them to pointers.

Custom Packed Messages

In many cases, a message must store a *non-linear* data structure using pointers. Examples of these are binary trees, hash tables etc. Thus, the message itself contains only a pointer to the actual data. When the message is sent to the same processor, these pointers point to the original locations, which are within the address space of the same processor. However, when such a message is sent to other processors, these pointers will point to invalid locations.

Thus, the programmer needs a way to “serialize” these messages *only if* the message crosses the address-space boundary. Charm++ provides a way to do this serialization by allowing the developer to override the default serialization methods generated by the Charm++ interface translator. Note that this low-level serialization has nothing to do with parameter marshalling or the PUP framework described later.

Packed messages are declared in the `.ci` file the same way as ordinary messages:

```
message PMessage;
```

Like all messages, the class PMessage needs to inherit from CMessage_PMessage and should provide two *static* methods: pack and unpack. These methods are called by the Charm++ runtime system, when the message is determined to be crossing address-space boundary. The prototypes for these methods are as follows:

```
static void *PMessage::pack(PMessage *in);
static PMessage *PMessage::unpack(void *in);
```

Typically, the following tasks are done in pack method:

- Determine size of the buffer needed to serialize message data.
- Allocate buffer using the CkAllocBuffer function. This function takes in two parameters: input message, and size of the buffer needed, and returns the buffer.
- Serialize message data into buffer (along with any control information needed to de-serialize it on the receiving side.
- Free resources occupied by message (including message itself.)

On the receiving processor, the unpack method is called. Typically, the following tasks are done in the unpack method:

- Allocate message using CkAllocBuffer function. *Do not use new to allocate message here. If the message constructor has to be called, it can be done using the in-place new operator.*
- De-serialize message data from input buffer into the allocated message.
- Free the input buffer using CkFreeMsg.

Here is an example of a packed-message implementation:

```
// File: pgm.ci
mainmodule PackExample {
    ...
    message PackedMessage;
    ...
};

// File: pgm.h
...
class PackedMessage : public CMessage_PackedMessage
{
public:
    BinaryTree<char> btree; // A non-linear data structure
    static void* pack(PackedMessage*);
    static PackedMessage* unpack(void*);
    ...
};
...

// File: pgm.C
...
void*
PackedMessage::pack(PackedMessage* inmsg)
{
    int treesize = inmsg->btree.getFlattenedSize();
    int totalsize = treesize + sizeof(int);
    char *buf = (char*)CkAllocBuffer(inmsg, totalsize);
    // buf is now just raw memory to store the data structure
```

(continues on next page)

(continued from previous page)

```

    int num_nodes = inmsg->btree.getNumNodes();
    memcpy(buf, &num_nodes, sizeof(int)); // copy numnodes into buffer
    buf = buf + sizeof(int);              // don't overwrite numnodes
    // copies into buffer, give size of buffer minus header
    inmsg->btree.Flatten((void*)buf, treesize);
    buf = buf - sizeof(int);              // don't lose numnodes
    delete inmsg;
    return (void*) buf;
}

PackedMessage*
PackedMessage::unpack(void* inbuf)
{
    // inbuf is the raw memory allocated and assigned in pack
    char* buf = (char*) inbuf;
    int num_nodes;
    memcpy(&num_nodes, buf, sizeof(int));
    buf = buf + sizeof(int);
    // allocate the message through Charm RTS
    PackedMessage* pmsg =
        (PackedMessage*) CkAllocBuffer(inbuf, sizeof(PackedMessage));
    // call "inplace" constructor of PackedMessage that calls constructor
    // of PackedMessage using the memory allocated by CkAllocBuffer,
    // takes a raw buffer inbuf, the number of nodes, and constructs the btree
    pmsg = new ((void*)pmsg) PackedMessage(buf, num_nodes);
    CkFreeMsg(inbuf);
    return pmsg;
}

...
PackedMessage* pm = new PackedMessage(); // just like always
pm->btree.Insert('A');
...

```

While serializing an arbitrary data structure into a flat buffer, one must be very wary of any possible alignment problems. Thus, if possible, the buffer itself should be declared to be a flat struct. This will allow the C++ compiler to ensure proper alignment of all its member fields.

Entry Method Attributes

Charm++ provides a handful of special attributes that entry methods may have. In order to give a particular entry method an attribute, you must specify the keyword for the desired attribute in the attribute list of that entry method's .ci file declaration. The syntax for this is as follows:

```
entry [attribute1, ..., attributeN] void EntryMethod(parameters);
```

Charm++ currently offers the following attributes that one may assign to an entry method: threaded, sync, exclusive, nokeep, notrace, appwork, immediate, expedited, inline, local, python, reductiontarget, aggregate.

threaded entry methods run in their own non-preemptible threads. These entry methods may perform blocking operations, such as calls to a sync entry method, or explicitly suspending themselves. For more details, refer to section 2.3.3.

sync entry methods are special in that calls to them are blocking—they do not return control to the caller until the method finishes execution completely. Sync methods may have return values; however, they may only return messages or data types that have the PUP method implemented. Callers must run in a thread separate from the

runtime scheduler, e.g. a threaded entry methods. Calls expecting a return value will receive it as the return from the proxy invocation:

```
ReturnMsg* m;  
m = A[i].foo(a, b, c);
```

For more details, refer to section 2.3.3.

exclusive entry methods should only exist on NodeGroup objects. One such entry method will not execute while some other exclusive entry methods belonging to the same NodeGroup object are executing on the same node. In other words, if one exclusive method of a NodeGroup object is executing on node N, and another one is scheduled to run on the same node, the second exclusive method will wait to execute until the first one finishes. An example can be found in `benchmarks/charm++/pingpong`.

nokeep entry methods take only a message as their lone argument, and the memory buffer for this message is managed by the Charm++ runtime system rather than by the user. This means that the user has to guarantee that the message will not be buffered for later usage or be freed in the user code. Additionally, users are not allowed to modify the contents of a nokeep message, since for a broadcast the same message can be reused for all entry method invocations on each PE. If a user frees the message or modifies its contents, a runtime error may result. An example can be found in `examples/charm++/histogram_group`.

notrace entry methods will not be traced during execution. As a result, they will not be considered and displayed in Projections for performance analysis. Additionally, `immediate` entry methods are by default `notrace` and will not be traced during execution.

appwork this entry method will be marked as executing application work. It will be used for performance analysis.

immediate entry methods are executed in an “immediate” fashion as they skip the message scheduling while other normal entry methods do not. Immediate entry methods can only be associated with NodeGroup objects, otherwise a compilation error will result. If the destination of such entry method is on the local node, then the method will be executed in the context of the regular PE regardless the execution mode of Charm++ runtime. However, in the SMP mode, if the destination of the method is on the remote node, then the method will be executed in the context of the communication thread. For that reason, immediate entry methods should be used for code that is performance critical and does not take too long in terms of execution time because long running entry methods can delay communication by occupying the communication thread for entry method execution rather than remote communication.

Such entry methods can be useful for implementing multicasts/reductions as well as data lookup when such operations are on the performance critical path. On a certain Charm++ PE, skipping the normal message scheduling prevents the execution of immediate entry methods from being delayed by entry functions that could take a long time to finish. Immediate entry methods are implicitly “exclusive” on each node, meaning that one execution of immediate message will not be interrupted by another. Function `CmiProbeImmediateMsg()` can be called in user codes to probe and process immediate messages periodically. Also note that `immediate` entry methods are by default `notrace` and are not traced during execution. An example of `immediate` entry method can be found in `examples/charm++/immediateEntryMethod`.

expedited entry methods skip the priority-based message queue in Charm++ runtime. It is useful for messages that require prompt processing when adding the `immediate` attribute to the message does not apply. Compared with the `immediate` attribute, the `expedited` attribute provides a more general solution that works for all types of Charm++ objects, i.e. Chare, Group, NodeGroup and Chare Array. However, expedited entry methods will still be scheduled in the lower-level Converse message queue, and be processed in the order of message arrival. Therefore, they may still suffer from delays caused by long running entry methods. An example can be found in `examples/charm++/satisfiability`.

inline entry methods will be called as a normal C++ member function if the message recipient happens to be on the same PE. The call to the function will happen inline, and control will return to the calling function after the inline method completes. Because of this, these entry methods need to be re-entrant as they could be called multiple times recursively. Parameters to the inline method will be passed by reference to avoid any copying, packing, or

unpacking of the parameters. This makes inline calls effective when large amounts of data are being passed, and copying or packing the data would be an expensive operation. Perfect forwarding has been implemented to allow for seamless passing of both lvalue and rvalue references. Note that calls with rvalue references must take place in the same translation unit as the `.decl.h` file to allow for the appropriate template instantiations. Alternatively, the method can be made templated and referenced from multiple translation units via `CK_TEMPLATES_ONLY`. An explicit instantiation of all lvalue references is provided for compatibility with existing code. If the recipient resides on a different PE, a regular message is sent with the message arguments packed up using PUP, and inline has no effect. An example “inlineem” can be found in `tests/charm++/megatest`.

local entry methods are equivalent to normal function calls: the entry method is always executed immediately. This feature is available only for Group objects and Chare Array objects. The user has to guarantee that the recipient chare element resides on the same PE. Otherwise, the application will abort with a failure. If the local entry method uses parameter marshalling, instead of marshalling input parameters into a message, it will pass them directly to the callee. This implies that the callee can modify the caller data if method parameters are passed by pointer or reference. The above description of perfect forwarding for inline entry methods also applies to local entry methods. Furthermore, input parameters are not required to be PUPable. Considering that these entry methods always execute immediately, they are allowed to have a non-void return value. An example can be found in `examples/charm++/hello/local`.

whenidle a local entry method meant to be used with `CkCallWhenIdle`, which registers an entry method to be called when a processor is idle. This mechanism provides a convenient way to do work (e.g. low priority or speculative) in the absence of other work. `whenidle` entry methods must return a `bool` value, indicating whether the entry method should be called when the processor is idle again, and accept a `double` argument representing the current timestamp. An example can be found in `examples/charm++/whenidle`.

python entry methods are enabled to be called from python scripts as explained in chapter 2.4.2. Note that the object owning the method must also be declared with the keyword `python`. Refer to chapter 2.4.2 for more details.

reductiontarget entry methods can be used as the target of reductions while taking arguments of the same type specified in the `contribute` call rather than a message of type `CkReductionMsg`. See section 2.2.3 for more information.

aggregate data sent to this entry method will be aggregated into larger messages before being sent, to reduce fine-grained overhead. The aggregation is handled by the Topological Routing and Aggregation Module (TRAM). The argument to this entry method must be a single PUPable object. More details on TRAM are given in the [TRAM section](#) of the libraries manual.

Controlling Delivery Order

By default, Charm++ processes the messages sent in roughly FIFO order when they arrive at a PE. For most programs, this behavior is fine. However, for optimal performance, some programs need more explicit control over the order in which messages are processed. Charm++ allows you to adjust delivery order on a per-message basis.

An example program demonstrating how to modify delivery order for messages and parameter marshaling can be found in `examples/charm++/prio`.

Queueing Strategies

The order in which messages are processed in the recipient’s queue can be set by explicitly setting the queueing strategy using one of the following constants. These constants can be applied when sending a message or invoking an entry method using parameter marshaling:

- `CK_QUEUEING_FIFO`: FIFO ordering
- `CK_QUEUEING_LIFO`: LIFO ordering
- `CK_QUEUEING_IFIFO`: FIFO ordering with *integer* priority

- CK_QUEUEING_ILIFO: LIFO ordering with *integer* priority
- CK_QUEUEING_BFIFO: FIFO ordering with *bitvector* priority
- CK_QUEUEING_BLIFO: LIFO ordering with *bitvector* priority
- CK_QUEUEING_LFIFO: FIFO ordering with *long integer* priority
- CK_QUEUEING_LLIFO: FIFO ordering with *long integer* priority

Parameter Marshaling

For parameter marshaling, the queueingtype can be set for CkEntryOptions, which is passed to an entry method invocation as the optional last parameter.

```
CkEntryOptions opts1, opts2;  
opts1.setQueueing(CK_QUEUEING_FIFO);  
opts2.setQueueing(CK_QUEUEING_LIFO);  
  
chare.entry_name(arg1, arg2, opts1);  
chare.entry_name(arg1, arg2, opts2);
```

When the message with opts1 arrives at its destination, it will be pushed onto the end of the message queue as usual. However, when the message with opts2 arrives, it will be pushed onto the *front* of the message queue.

Messages

For messages, the CkSetQueueing function can be used to change the order in which messages are processed, where queueingtype is one of the above constants.

```
void CkSetQueueing(MsgType message, int queueingtype)
```

The first two options, CK_QUEUEING_FIFO and CK_QUEUEING_LIFO, are used as follows:

```
MsgType *msg1 = new MsgType ;  
CkSetQueueing(msg1, CK_QUEUEING_FIFO);  
  
MsgType *msg2 = new MsgType ;  
CkSetQueueing(msg2, CK_QUEUEING_LIFO);
```

Similar to the parameter marshalled case described above, msg1 will be pushed onto the end of the message queue, while msg2 will be pushed onto the *front* of the message queue.

Prioritized Execution

The basic FIFO and LIFO strategies are sufficient to approximate parallel breadth-first and depth-first explorations of a problem space, but they do not allow more fine-grained control. To provide that degree of control, Charm++ also allows explicit prioritization of messages.

The other six queueing strategies involve the use of priorities . There are two kinds of priorities which can be attached to a message: *integer priorities* and *bitvector priorities* . These correspond to the *I* and *B* queueing strategies, respectively. In both cases, numerically lower priorities will be dequeued and delivered before numerically greater priorities. The FIFO and LIFO queueing strategies then control the relative order in which messages of the same priority will be delivered.

To attach a priority field to a message, one needs to set aside space in the message's buffer while allocating the message. To achieve this, the size of the priority field in bits should be specified as a placement argument to the new operator, as described in section 2.3.1. Although the size of the priority field is specified in bits, it is always padded to an integral number of ints. A pointer to the priority part of the message buffer can be obtained with this call: `void *CkPriorityPtr(MsgType msg).`

Integer priorities are quite straightforward. One allocates a message with an extra integer parameter to “new” (see the first line of the example below), which sets aside enough space (in bits) in the message to hold the priority. One then stores the priority in the message. Finally, one informs the system that the message contains an integer priority using `CkSetQueueing`:

```
MsgType *msg = new (8*sizeof(int)) MsgType;
*(int*)CkPriorityPtr(msg) = prio;
CkSetQueueing(msg, CK_QUEUEING_FIFO);
```

Bitvector Prioritization

Bitvector priorities are arbitrary-length bit-strings representing fixed-point numbers in the range 0 to 1. For example, the bit-string “001001” represents the number .001001. As with integer priorities, higher numbers represent lower priorities. However, bitvectors can be of arbitrary length, and hence the priority numbers they represent can be of arbitrary precision.

Arbitrary-precision priorities are often useful in AI search-tree applications. Suppose we have a heuristic suggesting that tree node N_1 should be searched before tree node N_2 . We therefore designate that node N_1 and its descendants will use high priorities, and that node N_2 and its descendants will use lower priorities. We have effectively split the range of possible priorities in two. If several such heuristics fire in sequence, we can easily split the priority range in two enough times that no significant bits remain, and the search begins to fail for lack of meaningful priorities to assign. The solution is to use arbitrary-precision priorities, i.e. bitvector priorities.

To assign a bitvector priority, two methods are available. The first is to obtain a pointer to the priority field using `CkPriorityPtr`, and then manually set the bits using the bit-setting operations inherent to C. To achieve this, one must know the format of the bitvector, which is as follows: the bitvector is represented as an array of unsigned integers. The most significant bit of the first integer contains the first bit of the bitvector. The remaining bits of the first integer contain the next 31 bits of the bitvector. Subsequent integers contain 32 bits each. If the size of the bitvector is not a multiple of 32, then the last integer contains 0 bits for padding in the least-significant bits of the integer.

The second way to assign priorities is only useful for those who are using the priority range-splitting described above. The root of the tree is assigned the null priority-string. Each child is assigned its parent's priority with some number of bits concatenated. The net effect is that the entire priority of a branch is within a small epsilon of the priority of its root.

It is possible to utilize unprioritized messages, integer priorities, and bitvector priorities in the same program. The messages will be processed in roughly the following order :

- Among messages enqueued with bitvector priorities, the messages are dequeued according to their priority. The priority “0000...” is dequeued first, and “1111...” is dequeued last.
- Unprioritized messages are treated as if they had the priority “1000...” (which is the “middle” priority, it lies exactly halfway between “0000...” and “1111...”).
- Integer priorities are converted to bitvector priorities. They are normalized so that the integer priority of zero is converted to “1000...” (the “middle” priority). To be more specific, the conversion is performed by adding 0x80000000 to the integer, and then treating the resulting 32-bit quantity as a 32-bit bitvector priority.
- Among messages with the same priority, messages are dequeued in FIFO order or LIFO order, depending upon which queuing strategy was used.

Additionally, long integer priorities can be specified by the *L* strategy.

A final reminder about prioritized execution: Charm++ processes messages in *roughly* the order you specify; it never guarantees that it will deliver the messages in *precisely* the order you specify. Thus, the correctness of your program should never depend on the order in which the runtime delivers messages. However, it makes a serious attempt to be “close”, so priorities can strongly affect the efficiency of your program.

Skiping the Queue

Some operations that one might want to perform are sufficiently latency-sensitive that they should never wait in line behind other messages. The Charm++ runtime offers two attributes for entry methods, expedited and immediate, to serve these needs. For more information on these attributes, see Section 2.3.1 and the example in `tests/charm++/megatest/immediatering.ci`.

Zero Copy Messaging API

Apart from using messages, Charm++ also provides APIs to avoid sender and receiver side copies. On RDMA enabled networks like GNI, Verbs, PAMI, OFI, and UCX these internally make use of one-sided communication by using the underlying Remote Direct Memory Access (RDMA) enabled network. For large arrays (few 100 KBs or more), the cost of copying during marshalling the message can be quite high. Using these APIs can help not only save the expensive copy operation but also reduce the application’s memory footprint by avoiding data duplication. Saving these costs for large arrays proves to be a significant optimization in achieving faster message send and receive times in addition to overall improvement in performance because of lower memory consumption. On the other hand, using these APIs for small arrays can lead to a drop in performance due to the overhead associated with one-sided communication. The overhead is mainly due to additional small messages required for sending the metadata message and the acknowledgment message on completion.

For within process data-transfers, this API uses regular memcpy to achieve zero copy semantics. Similarly, on CMA-enabled machines, in a few cases, this API takes advantage of CMA to perform inter-process intra-physical host data transfers. This API is also functional on non-RDMA enabled networks like regular ethernet, except that it does not avoid copies and behaves like a regular Charm++ entry method invocation.

There are three APIs that provide zero copy semantics in Charm++:

- Zero Copy Direct API
- Zero Copy Entry Method Send API
- Zero Copy Entry Method Post API

Zero Copy Direct API

The Zero Copy Direct API allows users to explicitly invoke a standard set of methods on predefined buffer information objects to avoid copies. Unlike the Entry Method API which calls the zero copy operation for every zero copy entry method invocation, the direct API provides a more flexible interface by allowing the user to exploit the persistent nature of iterative applications to perform zero copy operations using the same buffer information objects across iteration boundaries. It is also more beneficial than the Zero Copy Entry Method Send API because unlike the entry method Send API, which avoids just the sender side copy, the Zero Copy Direct API avoids both sender and receiver side copies.

To send an array using the Zero Copy Direct API, define a `CkNcpyBuffer` object on the sender chare specifying the pointer, size, a `CkCallback` object and optional `reg-mode` and `dereg-mode` parameters.


```
CkCallback srcCb(CkIndex_Ping1::sourceDone, thisProxy[thisIndex]);
// CkNcpyBuffer object representing the source buffer
CkNcpyBuffer source(arr1, arr1Size * sizeof(int), srcCb);
// ^ is equivalent to CkNcpyBuffer source(arr1, arr1Size * sizeof(int), srcCb, CK_
↳BUFFER_REG, CK_BUFFER_DEREG);
```

When used inside a CkNcpyBuffer object that represents the source buffer information, the callback is specified to notify about the safety of reusing the buffer and indicates that the get or put call has been completed. In those cases where the application can determine the safety of reusing the buffer through other synchronization mechanisms, the callback is not entirely useful and in such cases, CkCallback::ignore can be passed as the callback parameter. The optional reg-mode and dereg-mode parameters are used to determine the network registration mode and de-registration mode for the buffer. They are only relevant on networks requiring explicit memory registration for performing RDMA operations. These networks include GNI, OFI, UCX and Verbs. When the reg-mode is not specified by the user, the default reg-mode is considered to be CK_BUFFER_REG and similarly, when the dereg-mode is not specified by the user, the default dereg-mode is considered to be CK_BUFFER_DEREG.

Similarly, to receive an array using the Zero Copy Direct API, define another CkNcpyBuffer object on the receiver chare object specifying the pointer, the size, a CkCallback object and the optional reg-mode and dereg-mode parameters. When used inside a CkNcpyBuffer object that represents the destination buffer, the callback is specified to notify the completion of data transfer into the CkNcpyBuffer buffer. In those cases where the application can determine data transfer completion through other synchronization mechanisms, the callback is not entirely useful and in such cases, CkCallback::ignore can be passed as the callback parameter.

```
CkCallback destCb(CkIndex_Ping1::destinationDone, thisProxy[thisIndex]);
// CkNcpyBuffer object representing the destination buffer
CkNcpyBuffer dest(arr2, arr2Size * sizeof(int), destCb);
// is equivalent to CkNcpyBuffer dest(arr2, arr2Size * sizeof(int), destCb, CK_BUFFER_
↳REG, CK_BUFFER_DEREG);
```

Once the source CkNcpyBuffer and destination CkNcpyBuffer objects have been defined on the sender and receiver chares, to perform a get operation, send the source CkNcpyBuffer object to the receiver chare. This can be done using a regular entry method invocation as shown in the following code snippet, where the sender, arrProxy[0] sends its source object to the receiver chare, arrProxy[1].

```
// On Index 0 of arrProxy chare array
arrProxy[1].recvNcpySrcObj(source);
```

After receiving the sender's source CkNcpyBuffer object, the receiver can perform a get operation on its destination CkNcpyBuffer object by passing the source object as an argument to the runtime defined get method as shown in the following code snippet.

```
// On Index 1 of arrProxy chare array
// Call get on the local destination object passing the source object
dest.get(source);
```

This call performs a get operation, reading the remote source buffer into the local destination buffer.

Similarly, a receiver's destination CkNcpyBuffer object can be sent to the sender for the sender to perform a put on its source object by passing the source CkNcpyBuffer object as an argument to the runtime defined put method as shown in the code snippet.

```
// On Index 1 of arrProxy chare array
arrProxy[0].recvNcpyDestObj(dest);
```

```
// On Index 0 of arrProxy chare array
// Call put on the local source object passing the destination object
source.put(dest);
```

After the completion of either a get or a put, the callbacks specified in both the objects are invoked. Within the CkNcpyBuffer source callback, `sourceDone()`, the buffer can be safely modified or freed as shown in the following code snippet.

```
// Invoked by the runtime on source (Index 0)
void sourceDone() {
    // update the buffer to the next pointer
    updateBuffer();
}
```

Similarly, inside the CkNcpyBuffer destination callback, `destinationDone()`, the user is guaranteed that the data transfer is complete into the destination buffer and the user can begin operating on the newly available data as shown in the following code snippet.

```
// Invoked by the runtime on destination (Index 1)
void destinationDone() {
    // received data, begin computing
    computeValues();
}
```

The callback methods can also take a pointer to a CkDataMsg message. This message can be used to access the original buffer information object i.e. the CkNcpyBuffer objects used for the zero copy transfer. The buffer information object available in the callback allows access to all its information including the buffer pointer and the arbitrary reference pointer set using the method `setRef`. It is important to note that only the source CkNcpyBuffer object is accessible using the CkDataMsg in the source callback and similarly, the destination CkNcpyBuffer object is accessible using the CkDataMsg in the destination callback. The following code snippet illustrates the accessing of the original buffer pointer in the callback method by casting the data field of the CkDataMsg object into a CkNcpyBuffer object.

```
// Invoked by the runtime on source (Index 0)
void sourceDone(CkDataMsg *msg) {
    // Cast msg->data to a CkNcpyBuffer to get the source buffer information object
    CkNcpyBuffer *source = (CkNcpyBuffer *) (msg->data);

    // access buffer pointer and free it
    free(source->ptr);

    delete msg;
}
```

The following code snippet illustrates the usage of the `setRef` method.

```
const void *refPtr = &index;
CkNcpyBuffer source(arr1, arr1Size * sizeof(int), srcCb, CK_BUFFER_REG);
source.setRef(refPtr);
```

Similar to the buffer pointer, the user set arbitrary reference pointer can be also accessed in the callback method. This is shown in the next code snippet.

```
// Invoked by the runtime on source (Index 0)
void sourceDone(CkDataMsg *msg) {
    // update the buffer to the next pointer
```

(continues on next page)

(continued from previous page)

```

updateBuffer();

// Cast msg->data to a CkNcpyBuffer
CkNcpyBuffer *src = (CkNcpyBuffer *) (msg->data);

// access buffer pointer and free it
free(src->ptr);

// get reference pointer
const void *refPtr = src->ref;

delete msg;
}

```

The usage of `CkDataMsg` and `setRef` in order to access the original pointer and the arbitrary reference pointer is illustrated in `examples/charm++/zerocopy/direct_api/unreg/simple_get`

Both the source and destination buffers are of the same type i.e. `CkNcpyBuffer`. What distinguishes a source buffer from a destination buffer is the way the get or put call is made. A valid get call using two `CkNcpyBuffer` objects `obj1` and `obj2` is performed as `obj1.get(obj2)`, where `obj1` is the local destination buffer object and `obj2` is the remote source buffer object that was passed to this PE. Similarly, a valid put call using two `CkNcpyBuffer` objects `obj1` and `obj2` is performed as `obj1.put(obj2)`, where `obj1` is the local source buffer object and `obj2` is the remote destination buffer object that was passed to this PE.

In addition to the callbacks, the return values of get and put also indicate the completion of data transfer between the buffers. When the source and destination buffers are within the same process or on different processes within the same CMA-enabled physical node, the zerocopy data transfer happens immediately without an asynchronous RDMA call. In such cases, both the methods, get and put return an enum value of `CkNcpyStatus::complete`. This value of the API indicates the completion of the zerocopy data transfer. On the other hand, in the case of an asynchronous RDMA call, the data transfer is not immediate and the return enum value of the get and put methods is `CkNcpyStatus::incomplete`. This indicates that the data transfer is in-progress and not necessarily complete. Use of `CkNcpyStatus` in an application is illustrated in `examples/charm++/zerocopy/direct_api/reg/simple_get`.

Since callbacks in Charm++ allow to store a reference number, these callbacks passed into `CkNcpyBuffer` can be set with a reference number using the method `cb.setRefNum(num)`. Upon callback invocation, these reference numbers can be used to identify the buffers that were passed into the `CkNcpyBuffer` objects. Upon callback invocation, the reference number of the callback can be accessed using the `CkDataMsg` argument of the callback function. For a callback using a `CkDataMsg *msg`, the reference number is obtained by using the method `CkGetRefNum(msg)`. This is illustrated in `examples/charm++/zerocopy/direct_api/unreg/simple_get`. specifically useful where there is an indexed collection of buffers, where the reference number can be used to index the collection.

Note that the `CkNcpyBuffer` objects can be either statically declared or be dynamically allocated. Additionally, the objects are also reusable across iteration boundaries i.e. after sending the `CkNcpyBuffer` object, the remote PE can use the same object to perform get or put. This pattern of using the same objects across iterations is demonstrated in `benchmarks/charm++/zerocopy/direct_api/reg/pingpong`.

This API is demonstrated in `examples/charm++/zerocopy/direct_api`

Memory Registration and Registration Modes

There are four modes of memory registration for the Zero Copy API. These registration modes act as control switches on networks that require memory registration like GNI, OFI, UCX and Verbs, in order to perform RDMA operations. They dictate the functioning of the API providing flexible options based on user requirement. On other networks, where network memory management is not necessary (Netlrts) or is internally handled by the lower layer networking

API (PAMI, MPI), these switches are still supported to maintain API consistency by all behaving in the similar default mode of operation.

`CK_BUFFER_REG:`

`CK_BUFFER_REG` is the default mode that is used when no mode is passed. This mode doesn't distinguish between non-network and network data transfers. When this mode is passed, the buffer is registered immediately and this can be used for both non-network sends (`memcpy`) and network sends without requiring an extra message being sent by the runtime system for the latter case. This mode is demonstrated in `examples/charm++/zerocopy/direct_api/reg`

`CK_BUFFER_UNREG:`

When this mode is passed, the buffer is initially unregistered and it is registered only for network transfers where registration is absolutely required. For example, if the target buffer is on the same PE or same logical node (or process), since the `get` internally performs a `memcpy`, registration is avoided for non-network transfers. On the other hand, if the target buffer resides on a remote PE on a different logical node, the `get` is implemented through an RDMA call requiring registration. In such a case, there is a small message sent by the RTS to register and perform the RDMA operation. This mode is demonstrated in `examples/charm++/zerocopy/direct_api/unreg`

`CK_BUFFER_PREREG:`

This mode is only beneficial by implementations that use pre-registered memory pools. In Charm++, GNI and Verbs machine layers use pre-registered memory pools for avoiding registration costs. On other machine layers, this mode is supported, but it behaves similar to `CK_BUFFER_REG`. A custom allocator, `CkRdmaAlloc` can be used to allocate a buffer from a pool of pre-registered memory to avoid the expensive `malloc` and memory registration costs. For a buffer allocated through `CkRdmaAlloc`, the mode `CK_BUFFER_PREREG` should be passed to indicate the use of a mempooled buffer to the RTS. A buffer allocated with `CkRdmaAlloc` can be deallocated by calling a custom deallocator, `CkRdmaFree`. Although the allocator `CkRdmaAlloc` and deallocator, `CkRdmaFree` are beneficial on GNI and Verbs, the allocators are functional on other networks and allocate regular memory similar to a `malloc` call. Importantly, it should be noted that with the `CK_BUFFER_PREREG` mode, the allocated buffer's pointer should be used without any offsets. Using a buffer pointer with an offset will cause a segmentation fault. This mode is demonstrated in `examples/charm++/zerocopy/direct_api/prereg`

`CK_BUFFER_NOREG:`

This mode is used for just storing pointer information without any actual networking or registration information. It cannot be used for performing any zerocopy operations. This mode was added as it was useful for implementing a runtime system feature.

Memory De-registration and De-registration Modes

On network layers that require explicit memory registration, it is important to de-register the registered memory. Since networks are limited by the maximum registered or pinned memory, not de-registering pinned memory can lead to a pinned memory leak, which could potentially reduce application performance because it limits the amount of pinned memory available to the process.

Similar to memory registration modes, the ZC API also provides two de-registration modes.

`CK_BUFFER_DEREG:`

`CK_BUFFER_DEREG` is the default mode that is used when no dereg-mode is specified. This mode signals to the RTS that all the memory registered by the `CkNcopyBuffer` object should be de-registered by the RTS. Hence, when `CK_BUFFER_DEREG` is specified by the user, it is the runtime system's responsibility to de-register the buffer registered by the `CkNcopyBuffer` object. This dereg-mode is used when the user is not likely to reuse the buffer and wants the de-registration responsibility transferred to the RTS.

`CK_BUFFER_NODEREG:`

This mode is specified when the user doesn't want the RTS to de-register the buffer after the completion of the zero copy operation. It is beneficial when the user wants to reuse the same buffer (and reuse the `CkNcpyBuffer` object) by avoiding the cost of de-registration and registration.

Important Methods

The Zero Copy API provides important methods that offer utilities that can be used by the user.

The `deregisterMem` method is used by the user to de-register memory after the completion of the zero copy operations. This allows for other buffers to use the registered memory as machines/networks are limited by the maximum amount of registered or pinned memory. Registered memory can be de-registered by calling the `deregisterMem()` method on the `CkNcpyBuffer` object.

```
// de-register previously registered buffer
dest.deregisterMem();
```

In addition to `deregisterMem()`, there are other methods in a `CkNcpyBuffer` object that offer other utilities. The `init(const void *ptr, size_t cnt, CkCallback &cb, unsigned short int mode=CK_BUFFER_UNREG)` method can be used to re-initialize the `CkNcpyBuffer` object to new values similar to the ones that were passed in the constructor. For example, after using a `CkNcpyBuffer` object called `srcInfo`, the user can re-initialize the same object with other values. This is shown in the following code snippet.

```
// initialize src with new values
src.init(ptr, 200, newCb, CK_BUFFER_REG);
```

Additionally, the user can use another method `registerMem` in order to register a buffer that has been de-registered. Note that it is not required to call `registerMem` after a new initialization using `init` as `registerMem` is internally called on every new initialization. The usage of `registerMem` is illustrated in the following code snippet. Additionally, also note that following de-registration, if intended to register again, it is required to call `registerMem` even in the `CK_BUFFER_PREREG` mode when the buffer is allocated from a preregistered mempool. This is required to set the registration memory handles and will not incur any registration costs.

```
// register previously de-registered buffer
src.registerMem();
```

It should be noted that the Zero Copy Direct API is optimized only for point to point communication. Although it can be used for use cases where a single buffer needs to be broadcasted to multiple recipients, it is very likely that the API will currently perform suboptimally. This is primarily because of the implementation which assumes point to point communication for the Zero Copy Direct API. Having the same source buffer information (`CkNcpyBuffer`) being passed to a large number of processes, with each process performing a get call from the same source process will cause network congestion at the source process by slowing down each get operation because of competing get calls.

We are currently working on optimizing the Zero Copy Direct API for broadcast operations. Currently, the Zero Copy Entry Method Send API and the Zero Copy Entry Method Post API handle optimized broadcast operations by using a spanning tree as explained in the following sections.

Zero Copy Entry Method Send API

The Zero Copy Entry Method Send API allows the user to only avoid the sender side copy. On the receiver side, the buffer is allocated by the runtime system and a pointer to the Readonly buffer is provided to the user as an entry method parameter. For a broadcast operation, there is only one buffer allocated by the runtime system for each recipient process and a pointer to the same buffer is passed to all the recipient objects on that process (or logical node).

This API offloads the user from the responsibility of making additional calls to support zero copy semantics. It extends the capability of the existing entry methods with slight modifications in order to send large buffers without a copy.

To send an array using the Zero Copy Message Send API, specify the array parameter in the .ci file with the nocopy specifier.

```
// same .ci specification is used for p2p and bcast operations  
entry void foo (int size, nocopy int arr[size]);
```

While calling the entry method from the .C file, wrap the array i.e the pointer in a CkSendBuffer wrapper. The CkSendBuffer wrapper is essentially a method that constructs a CkNcopyBuffer around the passed pointer.

```
// for point to point send  
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr));  
  
// for readonly broadcast send  
arrayProxy.foo(500000, CkSendBuffer(arrPtr));
```

Until the RDMA operation is completed, it is not safe to modify the buffer. To be notified on completion of the RDMA operation, pass an optional callback object in the CkSendBuffer wrapper associated with the specific nocopy array.

```
CkCallback cb1(CkIndex_Foo::zerocopySent(), thisProxy[thisIndex]);  
  
// for point to point send  
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb1));  
  
// for readonly broadcast send  
arrayProxy.foo(500000, CkSendBuffer(arrPtr, cb1));
```

The CkSendBuffer wrapper also allows users to specify two optional mode parameters. These are used to determine the network registration mode and network de-registration mode for the buffer. It is only relevant on networks requiring explicit memory registration for performing RDMA operations. These networks include GNI, OFI, and Verbs. When the registration mode is not specified by the user, the default mode is considered to be CK_BUFFER_REG. Similarly, when the de-registration mode is not specified by the user, the default mode is considered to be CK_BUFFER_DEREG. Note that in order to specify a de-registration mode, the registration mode has to be specified in the constructor because of how default parameters work with constructors.

```
/* Specifying registration modes only, without any de-registration mode (uses default_  
↪de-reg mode) */  
  
// for point to point send  
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_REG));  
  
// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb1)); for REG reg-mode because_  
↪of REG being the default  
// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_UNREG)); for_  
↪UNREG reg-mode  
// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_PREREG)); for_  
↪PREREG reg-mode  
  
// for bcast send  
arrayProxy.foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_REG));  
  
// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, cb1)); for REG reg-mode because of_  
↪REG being the default  
// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_UNREG)); for UNREG_  
↪reg-mode
```

(continues on next page)

(continued from previous page)

```
// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_PREREG)); for PREREG_
↪reg-mode

/* Specifying de-registration modes */

// for point to point send
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_REG, CK_BUFFER_DEREG));

// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_REG)); for DERE_
↪dereg-mode because of DERE being the default
// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_REG, CK_BUFFER_
↪NODEREG)); for NODEREG dereg-mode

// for bcast send
arrayProxy.foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_REG, CK_BUFFER_DEREG));

// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_REG)); for DERE_
↪dereg-mode because of DERE being the default
// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, cb1, CK_BUFFER_REG, CK_BUFFER_
↪NODEREG)); for NODEREG dereg-mode
```

The memory registration mode and de-registration mode can also be specified without a callback, as shown below:

```
/* Specifying registration modes only, without any de-registration mode (uses default_
↪de-reg mode) */

// for point to point send
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_REG));

// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr)); for REG reg-mode because of_
↪REG being the default
// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_UNREG)); for UNREG reg-
↪mode
// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_PREREG)); for PREREG_
↪reg-mode

// for bcast send
arrayProxy.foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_REG));

// or arrayProxy.foo(500000, CkSendBuffer(arrPtr)); for REG reg-mode because of REG_
↪being the default
// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_UNREG)); for UNREG reg-
↪mode
// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_PREREG)); for PREREG reg-
↪mode

/* Specifying de-registration modes */

// for point to point send
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_REG, CK_BUFFER_DEREG));

// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_REG)); for DERE_
↪dereg-mode because of DERE being the default
// or arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_REG, CK_BUFFER_
↪NODEREG)); for NODEREG dereg-mode

// for bcast send
```

(continues on next page)

(continued from previous page)

```
arrayProxy.foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_REG, CK_BUFFER_DEREG));

// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_REG)); for DEREg dereg-
↪mode because of DEREg being the default
// or arrayProxy.foo(500000, CkSendBuffer(arrPtr, CK_BUFFER_REG, CK_BUFFER_NODEREg)); ↪
↪for NODEREg dereg-mode
```

In the case of point to point communication, the callback will be invoked on completion of the RDMA operation associated with the corresponding array.

For zero copy broadcast operations, the callback will be invoked when all the broadcast recipient processes have successfully received the broadcasted array. Note that this does not necessarily mean that the recipient objects have received the data i.e. it is possible that the process (or the logical node) has received the data, but the individual entry methods are still waiting in the scheduler (and are scheduled to run) and hence have not received the data.

Inside the callback, it is safe to overwrite the buffer sent via the Zero Copy Entry Method Send API.

```
//called when RDMA operation is completed
void zerocopySent() {
    // update the buffer to the next pointer
    updateBuffer();
}
```

The callback methods can also take a pointer to a CkDataMsg message. This message can be used to access the original buffer information object i.e. the CkNcpyBuffer object constructed by CkSendBuffer and used for the zero copy transfer. The buffer information object available in the callback allows access to all its information including the buffer pointer.

```
//called when RDMA operation is completed
void zerocopySent(CkDataMsg *msg)
{
    // Cast msg->data to a CkNcpyBuffer to get the source buffer information object
    CkNcpyBuffer *source = (CkNcpyBuffer *) (msg->data);

    // access buffer pointer and free it
    free(ptr); // if the callback is executed on the source process
    delete msg;
}
```

The RDMA call is associated with a nocopy array rather than the entry method. In the case of sending multiple nocopy arrays, each RDMA call is independent of the other. Hence, the callback applies to only the array it is attached to and not to all the nocopy arrays passed in an entry method invocation. On completion of the RDMA call for each array, the corresponding callback is separately invoked.

On the receiver side, the entry method is defined as a regular entry method in the .C file, with the nocopy parameter being received as a pointer as shown below:

```
void foo (int size, int *arr) {
    // data received in runtime system buffer pointed by arr
    // Note that 'arr' buffer is Readonly

    computeValues();
}
```

As an example, for an entry method with two nocopy array parameters, each called with the same callback, the callback will be invoked twice: on completing the transfer of each of the two nocopy parameters.

For multiple arrays to be sent via RDMA, declare the entry method in the .ci file as:


```
entry void foo (int size1, nocopy int arr1[size1], int size2, nocopy double_
↪arr2[size2]);
```

In the .C file, it is also possible to have different callbacks associated with each nocopy array.

```
CkCallback cb1(CkIndex_Foo::zerocopySent1(NULL), thisProxy[thisIndex]);
CkCallback cb2(CkIndex_Foo::zerocopySent2(NULL), thisProxy[thisIndex]);
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr1, cb1), 1024000, CkSendBuffer(arrPtr2,
↪cb2));
```

This API for point to point communication is demonstrated in `examples/charm++/zerocopy/entry_method_api` and `benchmarks/charm++/pingpong`. For broadcast operations, the usage of this API is demonstrated in `examples/charm++/zerocopy/entry_method_bcast_api`.

It should be noted that calls to entry methods with nocopy specified parameters is currently supported for point to point operations and only collection-wide broadcast operations like broadcasts across an entire chare array or group or nodegroup. It is yet to be supported for section broadcasts. Additionally, there is also no support for migration of chares that have pending RDMA transfer requests.

It should also be noted that the benefit of this API can be seen for large arrays on only RDMA enabled networks. On networks which do not support RDMA, and, for within process sends (which uses shared memory), the API is functional but doesn't show any performance benefit as it behaves like a regular entry method that copies its arguments.

Table 1 displays the message size thresholds for the Zero Copy Entry Method Send API on popular systems and build architectures. These results were obtained by running `benchmarks/charm++/zerocopy/entry_method_api/pingpong` in non-SMP mode on production builds. For message sizes greater than or equal to the displayed thresholds, the Zero Copy API is found to perform better than the regular message send API. For network layers that are not pamilrts, gni, verbs, ofi or mpi, the generic implementation is used.

1: Message Size Thresholds for which Zero Copy Entry API performs better than Regular API

Machine	Network	Build Architecture	Intra Proces- sor	Intra Host	Inter Host
Blue Gene/Q (Vesta)	PAMI	pamilrts-bluegeneq	4 MB	32 KB	256 KB
Cray XC30 (Edison)	Aries	gni-crayxc	1 MB	2 MB	2 MB
Cray XC30 (Edison)	Aries	mpi-crayxc	256 KB	8 KB	32 KB
Dell Cluster (Golub)	Infiniband	verbs-linux-x86_64	128 KB	2 MB	1 MB
Dell Cluster (Golub)	Infiniband	mpi-linux-x86_64	128 KB	32 KB	64 KB
Intel Cluster (Bridges)	Intel Omni-Path	ofi-linux-x86_64	64 KB	32 KB	32 KB
Intel KNL Cluster (Stam- pede2)	Intel Omni-Path	ofi-linux-x86_64	1 MB	64 KB	64 KB

Zero Copy Entry Method Post API

The Zero Copy Entry Method Post API is an extension of the Zero Copy Entry Method Send API. This API allows users to receive the sent data in a user posted buffer. In addition to saving the sender side copy, it also avoids the receiver side copy by not using any intermediate buffers and directly receiving the data in the user posted buffer. Unlike the Zero Copy Entry Method Send API, this API should be used when the user wishes to receive the data in a user posted buffer, which is allocated and managed by the user.

The posting of the receiver buffer happens at an object level, where each recipient object, for example, a chare array element or a group element or nodegroup element posts a receiver buffer using a special version of the entry method.

To send an array using the Zero Copy Post API, specify the array parameter in the .ci file with the `nocopypost` specifier.

```
// same .ci specification is used for p2p and bcast operations
entry void foo (int size, nocopypost int arr[size]);

// ^ note that the nocopypost specifier is different from nocopy and
// indicates the usage of the post API for the array arr
```

The sender side code for the Zero Copy Entry Method Post API is exactly the same as Zero Copy Entry Method Send API and can be referenced from the previous section. In this section, we will highlight the differences between the two APIs and demonstrate the usage of the Post API on the receiver side.

As previously mentioned, the Zero Copy Entry Method Post API posts user buffers to receive the data sent by the sender. This is done using a special overloaded version of the recipient entry method, called Post entry method. The overloaded function has all the entry method parameters and an additional `CkNcpyBufferPost` array parameter at the end of the signature. Additionally, the entry method parameters are specified as references instead of values. Inside this post entry method, the received references can be initialized by the user. The pointer reference is assigned to a user allocated buffer, i.e. the buffer in which the user wishes to receive the data. The size variable reference could be assigned to a value (or variable) that represents the size of the data of that type that needs to be received. If this reference variable is not assigned inside the post entry method, the size specified at the sender in the `CkSendBuffer` wrapper will be used as the default size. The post entry method also allows the receiver to specify the memory registration mode and the memory de-registration mode. This is done by indexing the `ncpyPost` array and assigning the `regMode` and `deregMode` parameters present inside each array element of the `CkNcpyBufferPost` array. When the network memory registration mode is unassigned by the user, the default `CK_BUFFER_REG` `regMode` is used. Similarly, when the de-registration mode is unassigned by the user, the default `CK_BUFFER_DEREG` `deregMode` is used.

For the entry method `foo` specified with a `nocopypost` specifier, the resulting post function defined in the `.C` file will be:

```
void foo (int &size, int *& arr, CkNcpyBufferPost *ncpyPost) {
    arr = myBuffer;           // myBuffer is a user allocated buffer

    size = 2000;              // 2000 ints need to be received.

    ncpyPost[0].regMode = CK_BUFFER_REG; // specify the regMode for the 0th pointer

    ncpyPost[0].deregMode = CK_BUFFER_DEREG; // specify the deregMode for the 0th_
    ↪pointer
}
```

In addition to the post entry method, the regular entry method also needs to be defined as in the case of the Entry Method Send API, where the `nocopypost` parameter is being received as a pointer as shown below:

```
void foo (int size, int *arr) {
    // arr points to myBuffer and the data sent is received in myBuffer
    // Note that 'arr' buffer is the same as myBuffer, which is user allocated and_
    ↪managed

    computeValues();
}
```

Similarly, for sending and receiving multiple arrays, the `.ci` file specification will be:

```
entry void foo(nocopypost int arr1[size1], int size1, nocopypost char arr2[size2],_
    ↪int size2);
```

In the `.C` file, we define a post entry method and a regular entry method:

```

// post entry method
void foo(int *& arr1, int & size1, char *& arr2, int & size2, CkNcpyBufferPost_
↳*ncpyPost) {

    arr1 = myBuffer1;
    ncpyPost[0].regMode = CK_BUFFER_UNREG;
    ncpyPost[0].deregMode = CK_BUFFER_DEREG;

    arr2 = myBuffer2; // myBuffer2 is allocated using CkRdmaAlloc
    ncpyPost[1].regMode = CK_BUFFER_PREREG;
    ncpyPost[1].deregMode = CK_BUFFER_NODEREG;
}

// regular entry method
void foo(int *arr1, int size1, char *arr2, int size2) {

    // sent buffer is received in myBuffer1, same as arr1

    // sent buffer is received in myBuffer2, same as arr2
}

```

It is important to note that the `CkNcpyBufferPost` array has as many elements as the number of `nocopypost` parameters in the entry method declaration in the `.ci` file. For `n` `nocopypost` parameters, the `CkNcpyBufferPost` array is indexed by 0 to `n-1`.

This API for point to point communication is demonstrated in `examples/charm++/zerocopy/entry_method_post_api` and for broadcast operations, the usage of this API is demonstrated in `examples/charm++/zerocopy/entry_method_bcast_post_api`.

Similar to the Zero Copy Entry Method Send API, it should be noted that calls to entry methods with `nocopypost` specified parameters are currently supported for point to point operations and only collection-wide broadcast operations like broadcasts across an entire chare array or group or nodegroup. It is yet to be supported for section broadcasts. Additionally, there is also no support for migration of chares that have pending RDMA transfer requests.

Using Zerocopy Post API in iterative applications

In iterative applications, when using the Zerocopy Post API, it is important to post different buffers for each iteration in order to receive each iteration's sent data in a separate buffer. Posting the same buffer across iterations could lead to overwriting the buffer when the receiver receives a message for that entry method. An example illustrating the use of the Zerocopy Post API to post different buffers in an iterative program, which uses SDAG is `examples/charm++/zerocopy/entry_method_post_api/reg/multiplePostedBuffers`. It is also important to post different buffers for each unique sender in order to receive each sender's data in a separate buffer. Posting the same buffer for different senders would overwrite the buffer when the receiver receives a message for that entry method.

Additionally, in iterative applications which use load balancing, for the load balancing iterations, it is required to ensure that the Zerocopy sends are performed only after all chare array elements have completed migration. This can be achieved by a chare array wide reduction after `ResumeFromSync` has been reached. This programming construct has the same effect as that of a barrier, enforced after array elements have completed load balancing. This is required to avoid cases where it is possible that the buffer is posted on receiving a send and is then freed by the destructor or the unpacking pup method, which is invoked by the RTS because of migration due to load balancing. This scheme of using a reduction following `ResumeFromSync` is illustrated in `examples/charm++/zerocopy/entry_method_post_api/reg/simpleZeroCopy` and `examples/charm++/zerocopy/entry_method_post_api/reg/multiplePostedBuffers`.

2.3.2 Callbacks

Callbacks provide a generic way to store the information required to invoke a communication target, such as a chare's entry method, at a future time. Callbacks are often encountered when writing library code, where they provide a simple way to transfer control back to a client after the library has finished. For example, after finishing a reduction, you may want the results passed to some chare's entry method. To do this, you would create an object of type `CkCallback` with the chare's `CkChareID` and entry method index, and pass this callback object to the reduction library.

Creating a `CkCallback` Object

There are several different types of `CkCallback` objects; the type of the callback specifies the intended behavior upon invocation of the callback. Callbacks must be invoked with the Charm++ message of the type specified when creating the callback. If the callback is being passed into a library which will return its result through the callback, it is the user's responsibility to ensure that the type of the message delivered by the library is the same as that specified in the callback. Messages delivered through a callback are not automatically freed by the Charm RTS. They should be freed or stored for future use by the user.

Callbacks that target chares require an "entry method index", an integer that identifies which entry method will be called. An entry method index is the Charm++ version of a function pointer. The entry method index can be obtained using the syntax:

```
int myIdx = CkIndex_ChareName::EntryMethod(parameters);
```

Here, `ChareName` is the name of the chare (group, or array) containing the desired entry method, `EntryMethod` is the name of that entry method, and `parameters` are the parameters taken by the method. These parameters are only used to resolve the proper `EntryMethod`; they are otherwise ignored.

Under most circumstances, entry methods to be invoked through a `CkCallback` must take a single message pointer as argument. As such, if the entry method specified in the callback is not overloaded, using `NULL` in place of parameters will suffice in fully specifying the intended target. If the entry method is overloaded, a message pointer of the appropriate type should be defined and passed in as a parameter when specifying the entry method. The pointer does not need to be initialized as the argument is only used to resolve the target entry method.

The intended behavior upon a callback's invocation is specified through the choice of callback constructor used when creating the callback. Possible constructors are:

1. `CkCallback(int ep, const CkChareID &id)` - When invoked, the callback will send its message to the given entry method (specified by the entry point index - `ep`) of the given Chare (specified by the chare id). Note that a chare proxy will also work in place of a chare id:

```
CkCallback(CkIndex_Foo::bar(NULL), thisProxy[thisIndex])
```

2. `CkCallback(void (*CallbackFn)(void *, void *), void *param)` - When invoked, the callback will pass `param` and the result message to the given C function, which should have a prototype like:

```
void myCallbackFn(void *param, void *message)
```

This function will be called on the processor where the callback was created, so `param` is allowed to point to heap-allocated data. Hence, this constructor should be used only when it is known that the callback target (which by definition here is just a C-like function) will be on the same processor as from where the constructor was called. Of course, you are required to free any storage referenced by `param`.

3. `CkCallback(CkCallback::ignore)` - When invoked, the callback will do nothing. This can be useful if a Charm++ library requires a callback, but you don't care when it finishes, or will find out some other way.
4. `CkCallback(CkCallback::ckExit)` - When invoked, the callback will call `CkExit()`, ending the Charm++ program.

5. `CkCallback(int ep, const CkArrayID &id)` - When invoked, the callback will broadcast its message to the given entry method of the given array. An array proxy will work in the place of an array id.
6. `CkCallback(int ep, const CkArrayIndex &idx, const CkArrayID &id)` - When invoked, the callback will send its message to the given entry method of the given array element.
7. `CkCallback(int ep, const CkGroupID &id)` - When invoked, the callback will broadcast its message to the given entry method of the given group.
8. `CkCallback(int ep, int onPE, const CkGroupID &id)` - When invoked, the callback will send its message to the given entry method of the given group member.
9. `CkCallback(CkFuture fut)` - When invoked, the callback will send its message to the given future. For a `ck::future` object, the underlying `CkFuture` is accesible via its `handle` method. For an example, see: `examples/charm++/hello/xarraySection/hello.C`

One final type of callback, `CkCallbackResumeThread()`, can only be used from within threaded entry methods. This callback type is discussed in section 2.3.2.

CkCallback Invocation

A properly initialized `CkCallback` object stores a global destination identifier, and as such can be freely copied, marshalled, and sent in messages. Invocation of a `CkCallback` is done by calling the function `send` on the callback with the result message as an argument. As an example, a library which accepts a `CkCallback` object from the user and then invokes it to return a result may have the following interface:

```
//Main library entry point, called by asynchronous users:
void myLibrary(...library parameters..., const CkCallback &cb)
{
    ..start some parallel computation, store cb to be passed to myLibraryDone later...
}

//Internal library routine, called when computation is done
void myLibraryDone(...parameters..., const CkCallback &cb)
{
    ...prepare a return message...
    cb.send(msg);
}
```

A `CkCallback` will accept any message type, or even `NULL`. The message is immediately sent to the user's client function or entry point. A library which returns its result through a callback should have a clearly documented return message type. The type of the message returned by the library must be the same as the type accepted by the entry method specified in the callback.

As an alternative to “send”, the callback can be used in a *contribute* collective operation. This will internally invoke the “send” method on the callback when the contribute operation has finished.

For examples of how to use the various callback types, please see `tests/charm++/megatest/callback.C`

Synchronous Execution with CkCallbackResumeThread

Threaded entry methods can be suspended and resumed through the `CkCallbackResumeThread` class. `CkCallbackResumeThread` is derived from `CkCallback` and has specific functionality for threads. This class automatically suspends the thread when the destructor of the callback is called. A suspended threaded client will resume when the “send” method is invoked on the associated callback. It can be used in situations when the return value is not needed, and only the synchronization is important. For example:

```
// Call the "doWork" method and wait until it has completed
void mainControlFlow() {
    ...perform some work...
    // call a library
    doWork(..., CkCallbackResumeThread());
    // or send a broadcast to a chare collection
    myProxy.doWork(..., CkCallbackResumeThread());
    // callback goes out of scope; the thread is suspended until doWork calls 'send' on
    ↪ the callback

    ...some more work...
}
```

Alternatively, if `doWork` returns a value of interest, this can be retrieved by passing a pointer to `CkCallbackResumeThread`. This pointer will be modified by `CkCallbackResumeThread` to point to the incoming message. Notice that the input pointer has to be cast to `(void*&)`:

```
// Call the "doWork" method and wait until it has completed
void mainControlFlow() {
    ...perform some work...
    MyMessage *mymsg;
    myProxy.doWork(..., CkCallbackResumeThread((void*&) mymsg));
    // The thread is suspended until doWork calls send on the callback

    ...some more work using "mymsg"...
}
```

Notice that the instance of `CkCallbackResumeThread` is constructed as an anonymous parameter to the “doWork” call. This insures that the callback is destroyed as soon as the function returns, thereby suspending the thread.

It is also possible to allocate a `CkCallbackResumeThread` on the heap or on the stack. We suggest that programmers avoid such usage, and favor the anonymous instance construction shown above. For completeness, we still present the code for heap and stack allocation of `CkCallbackResumeThread` callbacks below.

For heap allocation, the user must explicitly “delete” the callback in order to suspend the thread.

```
// Call the "doWork" method and wait until it has completed
void mainControlFlow() {
    ...perform some work...
    CkCallbackResumeThread cb = new CkCallbackResumeThread();
    myProxy.doWork(..., cb);
    ...do not suspend yet, continue some more work...
    delete cb;
    // The thread suspends now

    ...some more work after the thread resumes...
}
```

For a callback that is allocated on the stack, its destructor will be called only when the callback variable goes out of scope. In this situation, the function “`thread_delay`” can be invoked on the callback to force the thread to suspend. This also works for heap allocated callbacks.

```
// Call the "doWork" method and wait until it has completed
void mainControlFlow() {
    ...perform some work...
    CkCallbackResumeThread cb;
    myProxy.doWork(..., cb);
```

(continues on next page)

(continued from previous page)

```

...do not suspend yet, continue some more work...
cb.thread_delay();
// The thread suspends now

...some more work after the thread is resumed...
}

```

In all cases a *CkCallbackResumeThread* can be used to suspend a thread only once. (See `Main.cpp` of [Barnes-Hut MiniApp](#) for a complete example). *Deprecated usage*: in the past, “`thread_delay`” was used to retrieve the incoming message from the callback. While that is still allowed for backward compatibility, its usage is deprecated. The old usage is subject to memory leaks and dangling pointers.

Callbacks can also be tagged with reference numbers which can be matched inside SDAG code. When the callback is created, the creator can set the refnum and the runtime system will ensure that the message invoked on the callback’s destination will have that refnum. This allows the receiver of the final callback to match the messages based on the refnum value. (See `examples/charm++/examples/charm++/ckcallback` for a complete example).

2.3.3 Waiting for Completion

Threaded Entry Methods

Typically, entry methods run in the same thread of execution as the Charm++ scheduler. This prevents them from undertaking any actions that would cause their thread to block, as blocking would prevent the receiving and processing of incoming messages.

However, entry methods with the threaded attribute run in their own user-level nonpreemptible thread, and are therefore able to block without interrupting the runtime system. This allows them to undertake blocking operations or explicitly suspend themselves, which is necessary to use some Charm++ features, such as sync entry methods and futures.

For details on the threads API available to threaded entry methods, see chapter 3 of the Converse programming manual. The use of threaded entry methods is demonstrated in an example program located in `examples/charm++/threaded_ring`.

Sync Entry Methods

Generally, entry methods are invoked asynchronously and return `void`. Therefore, while an entry method may send data back to its invoker, it can only do so by invoking another asynchronous entry method on the chare object that invoked it.

However, it is possible to use sync entry methods, which have blocking semantics. The data returned by the invocation of such an entry method is available at the call site when it returns from blocking. This returned data can either be in the form of a Charm++ message or any type that has the PUP method implemented. Because the caller of a sync entry method will block, it must execute in a thread separate from the scheduler; that is, it must be a threaded entry method (*cf.* Section 2.3.3, above). If a sync entry method returns a value, it is provided as the return value from the invocation on the proxy object:

```

ReturnMsg* m;
m = A[i].foo(a, b, c);

```

An example of the use of sync entry methods is given in `tests/charm++/sync_square`.

Futures

Similar to Multilisp and other functional programming languages, Charm++ provides the abstraction of *futures*. In simple terms, a *future* is a contract with the runtime system to evaluate an expression asynchronously with the calling program. This mechanism promotes the evaluation of expressions in parallel as several threads concurrently evaluate the futures created by a program.

In some ways, a future resembles lazy evaluation. Each future is assigned to a particular thread (or to a chare, in Charm++) and, eventually, its value is delivered to the calling program. Once a future is created, a *reference* is returned immediately. However, if the *value* calculated by the future is needed, the calling program blocks until the value is available.

We provide both C-compatible and object-oriented interfaces for using futures, which include the following functions:

C	C++
<code>CkFuture CkCreateFuture(void)</code>	<code>ck::future()</code>
<code>void CkReleaseFuture(CkFuture fut)</code>	<code>void ck::future::release()</code>
<code>int CkProbeFuture(CkFuture fut)</code>	<code>bool ck::future::is_ready()</code>
<code>void *CkWaitFuture(CkFuture fut)</code>	<code>T ck::future::get()</code>
<code>void CkSendToFuture(CkFuture fut, void *msg)</code>	<code>void ck::future::set(T)</code>

The object-oriented versions are methods of `ck::future<T>`, which can be templated with any PUP-able type. Note, in most cases, messages/values cannot be retrieved via `get` when they were not been sent/set by a corresponding call to `set`; however, it can receive messages of supported, internal message types sent via `CkSendFuture`. Other message types must be wrapped as a PUP-able value and explicitly received as the expected message type(s); for example, one might wrap a message as `CkMarshallMsg` or `MsgPointerWrapper` then type-cast the (void*) message on the receiver-side. In such cases, one may consider using the C-like API for greater efficiency.

An example of the object-oriented interface is available under *examples/charm++/future*, with an equivalent example for the C-compatible interface presented below:

```
chare fib {
  entry fib(bool amIroot, int n, CkFuture f);
  entry [threaded] void run(bool amIroot, int n, CkFuture f);
};
```

```
void fib::run(bool amIroot, int n, CkFuture f) {
  if (n < THRESHOLD)
    result = seqFib(n);
  else {
    CkFuture f1 = CkCreateFuture();
    CkFuture f2 = CkCreateFuture();
    CProxy_fib::ckNew(0, n-1, f1);
    CProxy_fib::ckNew(0, n-2, f2);
    ValueMsg * m1 = (ValueMsg *) CkWaitFuture(f1);
    ValueMsg * m2 = (ValueMsg *) CkWaitFuture(f2);
    result = m1->value + m2->value;
    delete m1; delete m2;
  }
  if (amIroot) {
    CkPrintf("The requested Fibonacci number is : %d\n", result);
    CkExit();
  } else {
    ValueMsg *m = new ValueMsg();
    m->value = result;
    CkSendToFuture(f, m);
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

```

The constant *THRESHOLD* sets a limit value for computing the Fibonacci number with futures or just with the sequential procedure. Given value *n*, the program creates two futures using *CkCreateFuture*. Those futures are used to create two new chares that will carry out the computation. Next, the program blocks until the two component values of the recurrence have been evaluated. Function *CkWaitFuture* is used for that purpose. Finally, the program checks whether or not it is the root of the recursive evaluation. The very first chare created with a future is the root. If a chare is not the root, it must indicate that its future has finished computing the value. *CkSendToFuture* is meant to return the value for the current future.

Other functions complete the API for futures. *CkReleaseFuture* destroys a future. *CkProbeFuture* tests whether the future has already finished computing the value of the expression.

The maximum number of outstanding futures a PE may have is limited by the size of *CMK_REFNUM_TYPE*. Specifically, no more than $2^{SIZE} - 1$ futures, where *SIZE* is the size of *CMK_REFNUM_TYPE* in bits, may be outstanding at any time. Waiting on more futures will cause a fatal error in non-production builds, and will cause the program to hang in production builds. The default *CMK_REFNUM_TYPE* is `unsigned short`, limiting each PE to 65,535 outstanding futures. To increase this limit, build Charm++ with a larger *CMK_REFNUM_TYPE*, e.g. specifying `--with-refnum-type=uint` to use `unsigned int` when building Charm++.

There are additional facilities for operating on collections of futures, which include:

Function	Block- ing?	Description
<code>ck::wait_all</code>	Yes	Take a value, as soon as one is available, and return a pair with the value and position of the fulfilled future. (<code>std::pair<T, InputIter></code>)
<code>ck::wait_all</code>	Yes	Wait for all the futures to become available, and return a vector of values. (<code>std::vector<T></code>)
<code>ck::wait_some</code>	No	Take any immediately available values, returning the values and any outstanding futures. (<code>std::pair<std::vector<T>, InputIter></code>)

Note, these are also demonstrated in `examples/charm++/future`. The Converse version of future functions can be found in the [Futures](#) section.

Completion Detection

Completion detection is a method for automatically detecting completion of a distributed process within an application. This functionality is helpful when the exact number of messages expected by individual objects is not known. In such cases, the process must achieve global consensus as to the number of messages produced and the number of messages consumed. Completion is reached within a distributed process when the participating objects have produced and consumed an equal number of events globally. The number of global events that will be produced and consumed does not need to be known, just the number of producers is required.

The completion detection feature is implemented in Charm++ as a module, and therefore is only included when `-module completion` is specified when linking your application.

First, the detector should be constructed. This call would typically belong in application startup code (it initializes the group that keeps track of completion):

```
CProxy_CompletionDetector detector = CProxy_CompletionDetector::ckNew();
```

When it is time to start completion detection, invoke the following method of the library on *all* branches of the completion detection group:

```
void start_detection(int num_producers,
                    CkCallback start,
                    CkCallback all_produced,
                    CkCallback finish,
                    int prio);
```

The `num_producers` parameter is the number of objects (chares) that will produce elements. So if every chare array element will produce one event, then it would be the size of the array.

The `start` callback notifies your program that it is safe to begin producing and consuming (this state is reached when the module has finished its internal initialization).

The `all_produced` callback notifies your program when the client has called `done` with arguments summing to `num_producers`.

The `finish` callback is invoked when completion has been detected (all objects participating have produced and consumed an equal number of elements globally).

The `prio` parameter is the priority with which the completion detector will run. This feature is still under development, but it should be set below the application's priority if possible.

For example, the call

```
detector.start_detection(10,
                        CkCallback(CkIndex_chare1::start_test(), thisProxy),
                        CkCallback(CkIndex_chare1::produced_test(), thisProxy),
                        CkCallback(CkIndex_chare1::finish_test(), thisProxy),
                        0);
```

sets up completion detection for 10 producers. Once initialization is done, the callback associated with the `start_test` method will be invoked. Once all 10 producers have called `done` on the completion detector, the `produced_test` method will be invoked. Furthermore, when the system detects completion, the callback associated with `finish_test` will be invoked. Finally, the priority given to the completion detection library is set to 0 in this case.

Once initialization is complete (the “start” callback is triggered), make the following call to the library:

```
void CompletionDetector::produce(int events_produced)
void CompletionDetector::produce() // 1 by default
```

For example, within the code for a chare array object, you might make the following call:

```
detector.ckLocalBranch()->produce(4);
```

Once all the “events” that this chare is going to produce have been sent out, make the following call:

```
void CompletionDetector::done(int producers_done)
void CompletionDetector::done() // 1 by default
```

```
detector.ckLocalBranch()->done();
```

At the same time, objects can also consume produced elements, using the following calls:

```
void CompletionDetector::consume(int events_consumed)
void CompletionDetector::consume() // 1 by default
```

```
detector.ckLocalBranch()->consume();
```

Note that an object may interleave calls to `produce()` and `consume()`, i.e. it could produce a few elements, consume a few, etc. When it is done producing its elements, it should call `done()`, after which cannot `produce()` any more elements. However, it can continue to `consume()` elements even after calling `done()`. When the library detects that, globally, the number of produced elements equals the number of consumed elements, and all producers have finished producing (i.e. called `done()`), it will invoke the `finish` callback. Thereafter, `start_detection` can be called again to restart the process.

Quiescence Detection

In Charm++, quiescence is defined as the state in which no processor is executing an entry point, no messages are awaiting processing, and there are no messages in-flight. Charm++ provides two facilities for detecting quiescence: `CkStartQD` and `CkWaitQD`. `CkStartQD` registers with the system a callback that is to be invoked the next time quiescence is detected. Note that if immediate messages are used, QD cannot be used. `CkStartQD` has two variants which expect the following arguments:

1. A `CkCallback` object. The syntax of this call looks like:

```
CkStartQD(const CkCallback& cb);
```

Upon quiescence detection, the specified callback is called with no parameters. Note that using this variant, you could have your program terminate after quiescence is detected, by supplying the above method with a `CkExit` callback (Section 2.3.2).

2. An index corresponding to the entry function that is to be called, and a handle to the chare on which that entry function should be called. The syntax of this call looks like this:

```
CkStartQD(int Index, const CkChareID* chareID);
```

To retrieve the corresponding index of a particular entry method, you must use a static method contained within the (charmc-generated) `CkIndex` object corresponding to the chare containing that entry method. The syntax of this call is as follows:

```
myIdx=CkIndex_ChareClass::entryMethod(parameters);
```

where `ChareClass` is the C++ class of the chare containing the desired entry method, `entryMethod` is the name of that entry method, and `parameters` are the parameters taken by the method. These parameters are only used to resolve the proper `entryMethod`; they are otherwise ignored.

`CkWaitQD`, by contrast, does not register a callback. Rather, `CkWaitQD` *blocks* and does not return until quiescence is detected. It takes no parameters and returns no value. A call to `CkWaitQD` simply looks like this:

```
CkWaitQD();
```

Note that `CkWaitQD` should only be called from a threaded entry method because a call to `CkWaitQD` suspends the current thread of execution (cf. Section 2.3.3).

2.3.4 More Chare Array Features

The basic array features described previously (creation, messaging, broadcasts, and reductions) are needed in almost every Charm++ program. The more advanced techniques that follow are not universally needed, but represent many useful optimizations.

Local Access

It is possible to get direct access to a local array element using the proxy's `ckLocal` method, which returns an ordinary C++ pointer to the element if it exists on the local processor, and `NULL` if the element does not exist or is on another processor.

```
A1 *a=a1[i].ckLocal();  
if (a==NULL) // ...is remote -- send message  
else // ...is local -- directly use members and methods of a
```

Note that if the element migrates or is deleted, any pointers obtained with `ckLocal` are no longer valid. It is best, then, to either avoid `ckLocal` or else call `ckLocal` each time the element may have migrated; e.g., at the start of each entry method.

An example of this usage is available in `examples/charm++/topology/matmul3d`.

Advanced Array Creation

There are several ways to control the array creation process. You can adjust the map and bindings before creation, change the way the initial array elements are created, create elements explicitly during the computation, and create elements implicitly, “on demand”.

You can create all of an arrays elements using any one of these methods, or create different elements using different methods. An array element has the same syntax and semantics no matter how it was created.

Configuring Array Characteristics Using CkArrayOptions

The array creation method `ckNew` actually takes a parameter of type `CkArrayOptions`. This object describes several optional attributes of the new array.

The most common form of `CkArrayOptions` is to set the number of initial array elements. A `CkArrayOptions` object will be constructed automatically in this special common case. Thus the following code segments all do exactly the same thing:

```
// Implicit CkArrayOptions  
a1=CProxy_A1::ckNew(parameters,nElements);  
  
// Explicit CkArrayOptions  
a1=CProxy_A1::ckNew(parameters,CkArrayOptions(nElements));  
  
// Separate CkArrayOptions  
CkArrayOptions opts(nElements);  
a1=CProxy_A1::ckNew(parameters,opts);
```

Note that the “numElements” in an array element is simply the `numElements` passed in when the array was created. The true number of array elements may grow or shrink during the course of the computation, so `numElements` can become out of date. This “bulk” constructor approach should be preferred where possible, especially for large arrays. Bulk construction is handled via a broadcast which will be significantly more efficient in the number of messages required than inserting each element individually, which will require one message send per element.

Examples of bulk construction are commonplace, see `examples/charm++/jacobi3d-sdag` for a demonstration of the slightly more complicated case of multidimensional chare array bulk construction.

`CkArrayOptions` can also be used for bulk creation of sparse arrays when the sparsity of the array can be described in terms of a start index, an end index, and a step index. The start, end, and step can either be passed into the

CkArrayOptions constructor, or set one at a time. The following shows two different ways to create CkArrayOptions for a 2D array with only the odd indices from (1,1) to (10,10) being populated:

```
// Set at construction
CkArrayOptions options(CkArrayIndex2D(1,1),
                      CkArrayIndex2D(10,10),
                      CkArrayIndex(2,2));

// Set one at a time
CkArrayOptions options;
options.setStart(CkArrayIndex2D(1,1))
        .setEnd(CkArrayIndex2D(10,10))
        .setStep(CkArrayIndex2D(2,2));
```

The default for start is 0^d and the default for step is 1^d (where d is the dimension of the array), so the following are equivalent:

```
// Specify just the number of elements
CkArrayOptions options(nElements);

// Specify just the end index
CkArrayOptions options;
options.setEnd(CkArrayIndex1D(nElements));

// Specify all three indices
CkArrayOptions options;
options.setStart(CkArrayIndex1D(0))
        .setEnd(CkArrayIndex1D(nElements))
        .setStep(CkArrayIndex1D(1));
```

In addition to controlling how many elements and at which indices to create them, CkArrayOptions contains a few flags that the runtime can use to optimize handling of a given array. If the array elements will only migrate at controlled points (such as periodic load balancing with `AtASync()`), this is signaled to the runtime by calling `opts.setAnytimeMigration(false)`¹¹. If all array elements will be inserted by bulk creation or by `fooArray[x].insert()` calls, signal this by calling `opts.setStaticInsertion(true)`¹².

Initial Placement Using Map Objects

You can use CkArrayOptions to specify a “map object” for an array. The map object is used by the array manager to determine the “home” PE of each element. The home PE is the PE upon which it is initially placed, which will retain responsibility for maintaining the location of the element.

There is a default map object, which maps 1D array indices in a block fashion to processors, and maps other array indices based on a hash function. Some other mappings such as round-robin (RRMap) also exist, which can be used similar to custom ones described below.

A custom map object is implemented as a group which inherits from CkArrayMap and defines these virtual methods:

```
class CkArrayMap : public Group {
public:
    // ...

    // Return an arrayHdl, given some information about the array
    virtual int registerArray(CkArrayIndex& numElements, CkArrayID aid);
```

(continues on next page)

¹¹ At present, this optimizes broadcasts to not save old messages for immigrating chares.

¹² This can enable a slightly faster default mapping scheme.

(continued from previous page)

```
// Return the home processor number for this element of this array
virtual int procNum(int arrayHdl, const CkArrayIndex &element);
};
```

For example, a simple 1D blockmapping scheme. Actual mapping is handled in the `procNum` function.

```
// In the .ci file:
group BlockMap : CkArrayMap {
    entry BlockMap();
};

// In the .C/.h files
class BlockMap : public CkArrayMap {
public:
    BlockMap(void) {}
    BlockMap(CkMigrateMessage* m) {}
    int registerArray(CkArrayIndex& numElements, CkArrayID aid) {
        return 0;
    }
    int procNum(int /*arrayHdl*/, const CkArrayIndex &idx) {
        int elem = *(int*)idx.data();
        int penum = (elem / (32 / CkNumPes()));
        return penum;
    }
};
```

Note that the first argument to the `procNum` method exists for reasons internal to the runtime system and is not used in the calculation of processor numbers.

Once you've instantiated a custom map object, you can use it to control the location of a new array's elements using the `setMap` method of the `CkArrayOptions` object described above. For example, if you've declared a map object named `BlockMap`:

```
// Create the map group
CProxy_BlockMap myMap=CProxy_BlockMap::ckNew();

// Make a new array using that map
CkArrayOptions opts(nElements);
opts.setMap(myMap);
a1=CProxy_A1::ckNew(parameters,opts);
```

A very basic example which also demonstrates how initial elements are created may be found in `examples/charm++/array_map`

An example which constructs one element per physical node may be found in `examples/charm++/PUP/pupDisk`.

Other 3D Torus network oriented map examples are in `examples/charm++/topology`.

Initial Elements

The map object described above can also be used to create the initial set of array elements in a distributed fashion. An array's initial elements are created by its map object, by making a call to `populateInitial` on each processor. This function is defined in the `CkArrayMap` class to iterate through the index space of the initial elements (defined as a start index, end index, and step index) and call `procNum` for each index. If the PE returned by `procNum` is the same as the calling PE, then an object is created on that PE.

If there is a more direct way to determine the elements to create on each PE, the `populateInitial` function can be overridden by using the following signature:

```
virtual void populateInitial(int arrayHdl, CkArrayOptions& options,
    void* ctorMsg, CkArray* mgr)
```

In this call, `arrayHdl` is the value returned by `registerArray`, `options` contains the `CkArrayOptions` passed into the array at construction, `ctorMsg` is the constructor message to pass, and `mgr` is the array manager which creates the elements.

To create an element, call `void CkArray::insertInitial(CkArrayIndex idx, void* ctorMsg)` on `mgr`, passing in the index and a copy of the constructor message. For example, to insert a 2D element (x,y), call:

```
mgr->insertInitial(CkArrayIndex2D(x,y), CkCopyMsg(&msg));
```

After inserting elements, inform the array manager that all elements have been created, and free the constructor message:

```
mgr->doneInserting();
CkFreeMsg(msg);
```

A simple example using `populateInitial` can be found in `examples/charm++/array_map`

Bound Arrays

You can “bind” a new array to an existing array using the `bindTo` method of `CkArrayOptions`. Bound arrays act like separate arrays in all ways except for migration- corresponding elements of bound arrays always migrate together. For example, this code creates two arrays A and B which are bound together- A[i] and B[i] will always be on the same processor.

```
// Create the first array normally
aProxy=CProxy_A::ckNew(parameters,nElements);
// Create the second array bound to the first
CkArrayOptions opts(nElements);
opts.bindTo(aProxy);
bProxy=CProxy_B::ckNew(parameters,opts);
```

An arbitrary number of arrays can be bound together- in the example above, we could create yet another array C and bind it to A or B. The result would be the same in either case- A[i], B[i], and C[i] will always be on the same processor.

There is no relationship between the types of bound arrays- it is permissible to bind arrays of different types or of the same type. It is also permissible to have different numbers of elements in the arrays, although elements of A which have no corresponding element in B obey no special semantics. Any method may be used to create the elements of any bound array.

Bound arrays are often useful if A[i] and B[i] perform different aspects of the same computation, and thus will run most efficiently if they lie on the same processor. Bound array elements are guaranteed to always be able to interact using `ckLocal` (see section 2.3.4), although the local pointer must be refreshed after any migration. This should be done during the `pup` routine. When migrated, all elements that are bound together will be created at the new processor before `pup` is called on any of them, ensuring that a valid local pointer to any of the bound objects can be obtained during the `pup` routine of any of the others.

For example, an array *Alibrary* is implemented as a library module. It implements a certain functionality by operating on a data array *dest* which is just a pointer to some user provided data. A user defined array *UserArray* is created and bound to the array *Alibrary* to take advantage of the functionality provided by the library. When bound array element migrated, the *data* pointer in *UserArray* is re-allocated in `pup()`, thus *UserArray* is responsible to refresh the pointer *dest* stored in *Alibrary*.

```

class Alibrary: public CProxy_Alibrary {
public:
    ...
    void set_ptr(double *ptr) { dest = ptr; }
    virtual void pup(PUP::er &p);
private:
    double *dest; // point to user data in user defined bound array
};

class UserArray: public CProxy_UserArray {
public:
    virtual void pup(PUP::er &p) {
        p|len;
        if(p.isUnpacking()) {
            data = new double[len];
            Alibrary *myfellow = AlibraryProxy(thisIndex).ckLocal();
            myfellow->set_ptr(data); // refresh data in bound array
        }
        p(data, len);
    }
private:
    CProxy_Alibrary AlibraryProxy; // proxy to my bound array
    double *data; // user allocated data pointer
    int len;
};

```

A demonstration of bound arrays can be found in `tests/charm++/startupTest`

Note that if any bound array element sets `usesAtSync=true` in its constructor, then users must ensure that `AtSync()` is called on all of those array elements. If a bound array element does not have the `usesAtSync` flag set to true, then it will migrate along with any elements it is bound to when they migrate. In this case, such an array element does not need to call `AtSync()` itself.

Dynamic Insertion

In addition to creating initial array elements using `ckNew(...)`, you can also create array elements during the computation.

You insert elements into the array by indexing the proxy and calling `insert(...)`. This call optionally takes parameters, which are passed to the constructor, and a processor number, where the element will be created. Array elements can be inserted in any order from any processor at any time. Array elements need not be contiguous.

If using `insert` to create all the elements of the array, you must call `CProxy_Array::doneInserting()` before using the array.

```

// In the .C file:
int x,y,z;
CProxy_A1 a1=CProxy_A1::ckNew(); // Creates a new, empty 1D array
for (x=...) {
    a1[x].insert(parameters); // Bracket syntax
    a1(x+1).insert(parameters); // or equivalent parenthesis syntax
}
a1.doneInserting();

CProxy_A2 a2=CProxy_A2::ckNew(); // Creates 2D array
for (x=...) for (y=...)

```

(continues on next page)

(continued from previous page)

```

    a2(x,y).insert(parameters); // Can't use brackets!
a2.doneInserting();

CProxy_A3 a3=CProxy_A3::ckNew(); // Creates 3D array
for (x=...) for (y=...) for (z=...)
    a3(x,y,z).insert(parameters);
a3.doneInserting();

CProxy_AF aF=CProxy_AF::ckNew(); // Creates user-defined index array
for (...) {
    aF[CkArrayIndexFoo(...)].insert(parameters); // Use brackets...
    aF(CkArrayIndexFoo(...)).insert(parameters); // ...or parenthesis
}
aF.doneInserting();

```

The `doneInserting()` call starts the reduction manager (see “Array Reductions”) and load balancer (see 2.2.6). Since these objects need to know about all the array’s elements, they must be started after the initial elements are inserted. You may call `doneInserting()` multiple times, but only the first call actually does anything. You may even insert or destroy elements after a call to `doneInserting()`, with different semantics - see the reduction manager. For `AtSync` load balancing, subsequent dynamic insertion or deletion sessions should begin with a call to `CProxy_Array::startInserting()` and end with a call to `doneInserting()`. `startInserting()` is also idempotent and can be called multiple times with only the first having any effect until `doneInserting()` is called on the same array proxy on the same PE.

If you do not specify one, the system will choose a processor to create an array element on based on the current map object.

A demonstration of dynamic insertion is available: `examples/charm++/hello/fancyarray`

Demand Creation

Demand Creation is a specialized form of dynamic insertion. Normally, invoking an entry method on a nonexistent array element is an error. But if you add the attribute `[createhere]` or `[createhome]` to an entry method, the array manager will “demand create” a new element to handle the message.

With `[createhome]`, the new element will be created on the home processor, which is most efficient when messages for the element may arrive from anywhere in the machine. With `[createhere]`, the new element is created on the sending processor, which is most efficient if when messages will often be sent from that same processor.

The new element is created by calling its default (taking no parameters) constructor, which must exist and be listed in the `.ci` file. A single array can have a mix of demand-creation and classic entry methods; and demand-created and normally created elements.

A simple example of demand creation `tests/charm++/demand_creation`.

Asynchronous Array Creation

Normally, `CProxy_Array::ckNew` call must always be made from PE 0. However, asynchronous array creation can be used to lift this restriction and let the array creation be made from any PE. To do this, `CkCallback` must be given as an argument for `ckNew` to provide the created char array’s `CkArrayID` to the callback function.

```

CProxy_SomeProxy::ckNew(parameters, nElements, CkCallback(CkIndex_
↪MyClass::someFunction(NULL), thisProxy));

```

(continues on next page)

(continued from previous page)

```
void someFunction(CkArrayCreatedMsg *m) {
    CProxy_AnotherProxy(m->aid)[index].myFunction(); // m->aid is CkArrayID
    delete m;
}
```

Similar to the standard array creation method, arguments to the array element constructor calls are taken first, followed by the dimensions of the array. Note that the parameters field can be optional, if the default constructor is expected to be used.

Alternatively, CkArrayOptions can be used in place of nElements to further configure array characteristics.

```
// Creating a 3-dimensional chare array with 2 parameters
CkArrayOptions options(dimX, dimY, dimZ);
CProxy_AnotherProxy::ckNew(param1, param2, options, CkCallback(CkIndex_
↪SomeClass::anotherFunction(NULL), thisProxy));
```

A demonstration of asynchronous array creation can be found in `examples/charm++/hello/darray`.

User-defined Array Indices

Charm++ array indices are arbitrary collections of integers. To define a new array index type, you create an ordinary C++ class which inherits from CkArrayIndex, allocates custom data in the space it has set aside for index data, and sets the “nInts” member to the length, in integers, of the custom index data.

For example, if you have a structure or class named “Foo”, you can use a Foo object as an array index by defining the class:

```
// Include to inherit from CkArrayIndex
#include <charm++.h>

class CkArrayIndexFoo : public CkArrayIndex {
private:
    Foo* f;
public:
    CkArrayIndexFoo(const Foo &in) {
        f = new (index) Foo(in);
        nInts = sizeof(Foo)/sizeof(int);
    }
};
```

Note that Foo must be allocated using placement new pointing to the “index” member of CkArrayIndex. Furthermore, its size must be an integral number of integers- you must pad it with zero bytes if this is not the case. Also, Foo must be a simple class- it cannot contain pointers, have virtual functions, or require a destructor. Finally, there is a Charm++ configuration-time option called CK_ARRAYINDEX_MAXLEN which is the largest allowable number of integers in an array index. The default is 3; but you may override this to any value by passing “-DCK_ARRAYINDEX_MAXLEN=n” to the Charm++ build script as well as all user code. Larger values will increase the size of each message.

You can then declare an array indexed by Foo objects with

```
// in the .ci file:
array [Foo] AF { entry AF(); ... }

// in the .h file:
class AF : public CBase_AF
```

(continues on next page)

(continued from previous page)

```
{ public: AF() {} ... }

// in the .C file:
Foo f;
CProxy_AF a=CProxy_AF::ckNew();
a[CkArrayIndexFoo(f)].insert();
...
```

Note that since our CkArrayIndexFoo constructor is not declared with the explicit keyword, we can equivalently write the last line as:

```
a[f].insert();
```

The array index (an object of type Foo) is then accessible as “thisIndex”. For example:

```
// in the .C file:
AF::AF() {
    Foo myF=thisIndex;
    functionTakingFoo(myF);
}
```

A demonstration of user defined indices can be seen in `examples/charm++/hello/fancyarray`.

2.3.5 Sections: Subsets of a Chare Array/Group

Charm++ supports defining and communicating with subsets of a chare array or group. This entity is called a chare array section or a group section (*section*). Section elements are addressed via a section proxy. Charm++ also supports sections which are a subset of elements of multiple chare arrays/groups of the same type (see 2.3.5).

Multicast operations (a broadcast to all members of a section) are directly supported by the section proxy. For array sections, multicast operations by default use optimized spanning trees via the CkMulticast library in Charm++. For group sections, multicast operations by default use an unoptimized direct-sending implementation. To optimize messaging, group sections need to be manually delegated to CkMulticast (see 2.3.5). Reductions are also supported for both arrays and group sections via the CkMulticast library.

Array and group sections work in mostly the same way. Check `examples/charm++/groupsection` for a group section example and `examples/charm++/arraysection` for an array section example.

Section Creation

Array sections

For each chare array “A” declared in a ci file, a section proxy of type “CProxySection_A” is automatically generated in the decl and def header files. You can create an array section in your application by invoking `ckNew()` function of the CProxySection. The user will need to provide array indexes of all the array section members through either explicit enumeration, or an index range expression. For example, for a 3D array:

```
std::vector<CkArrayIndex3D> elems; // add array indices
for (int i=0; i<10; i++)
    for (int j=0; j<20; j+=2)
        for (int k=0; k<30; k+=2)
            elems.emplace_back(i, j, k);
CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, elems);
```

Alternatively, one can do the same thing by providing the index range [lbound:ubound:stride] for each dimension:

```
CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, 0, 9, 1, 0, 19, 2, 0, 29, 2);
```

The above code creates a section proxy that contains array elements [0:9, 0:19:2, 0:29:2].

For user-defined array index other than CkArrayIndex1D to CkArrayIndex6D, one needs to use the generic array index type: CkArrayIndex.

```
std::vector<CkArrayIndex> elems; // add array indices
CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, elems);
```

Group sections

Group sections are created in the same way as array sections. A group “A” will have an associated “CProxySection_A” type which is used to create a section and obtain a proxy. In this case, `ckNew()` will receive the list of PE IDs which will form the section. See `examples/charm++/groupsection` for an example.

Important: It is important to note that Charm++ does not automatically delegate group sections to the internal CkMulticast library, and instead defaults to a point-to-point implementation of multicasts. To use CkMulticast with group sections, the user must manually delegate after invoking group creation. See 2.3.5 for information on how to do this.

Creation order restrictions

Attention: Array sections should be created in post-constructor entry methods to avoid race conditions.

If the user wants to invoke section creation from a group, special care must be taken that the collection for which we are creating a section (array or group) already exists.

For example, suppose a user wants to create a section of array “A” from an entry method in group “G”. Because groups are created before arrays in Charm++, and there is no guarantee of creation order of groups, there is a risk that array A’s internal structures have not been initialized yet on every PE, causing section creation to fail. As such, the application must ensure that A has been created before attempting to create a section.

If the section is created from inside an array element there is no such risk.

Section Multicasts

Once the proxy is obtained at section creation time, the user can broadcast to all the section members, like this:

```
CProxySection_Hello proxy;
proxy.someEntry(...); // section broadcast
```

See `examples/charm++/arraysection` for examples on how sections are used.

You can send the section proxy in a message to another processor, and still safely invoke the entry functions on the section proxy.

Optimized multicast via CkMulticast

Charm++ has a built-in CkMulticast library that optimizes section communications. By default, the Charm++ runtime system will use this library for array and cross-array sections. For group sections, the user must manually delegate the section proxy to CkMulticast (see 2.3.5).

By default, CkMulticast builds a spanning tree for multicast/reduction with a factor of 2 (binary tree). One can specify a different branching factor when creating the section.

```
CProxySection_Hello sectProxy = CProxySection_Hello::ckNew(..., 3); // factor is 3
```

Note that, to use CkMulticast library, all multicast messages must inherit from CkMcastBaseMsg, as the following example shows.

```
class HiMsg : public CkMcastBaseMsg, public CMessage_HiMsg
{
public:
    int *data;
};
```

Attention: CkMcastBaseMsg must come first, this is important for CkMulticast library to retrieve section information from the message.

Due to this restriction, when using CkMulticast you must define messages explicitly for multicast entry functions and no parameter marshalling can be used.

Section Reductions

Reductions over the elements of a section are supported through the CkMulticast library. As such, to perform reductions, the section must have been delegated to CkMulticast, either automatically (which is the default case for array sections), or manually for group sections.

Since an array element can be a member of multiple array sections, it is necessary to disambiguate between which array section reduction it is participating in each time it contributes to one. For this purpose, a data structure called CkSectionInfo is created by CkMulticast library for each array section that the array element belongs to. During a section reduction, the array element must pass the CkSectionInfo as a parameter in the `contribute()`. The CkSectionInfo for a section can be retrieved from a message in a multicast entry point using function call `CkGetSectionInfo()`:

```
CkSectionInfo cookie;

void SayHi(HiMsg *msg)
{
    CkGetSectionInfo(cookie, msg); // update section cookie every time
    int data = thisIndex;
    CProxySection_Hello::contribute(sizeof(int), &data, CkReduction::sum_int, cookie,
    ↪cb);
}
```

Note that the cookie cannot be used as a one-time local variable in the function, the same cookie is needed for the next contribute. This is because the cookie includes some context-sensitive information (e.g., the reduction counter). Subsequent invocations of `CkGetSectionInfo()` only updates part of the data in the cookie, rather than creating a brand new one.

Similar to array reductions, to use section-based reductions, a reduction client CkCallback object must be created. You may pass the client callback as an additional parameter to contribute. If different contribute calls to the same reduction operation pass different callbacks, some (unspecified, unreliable) callback will be chosen for use.

See the following example:

```
CkCallback cb(CkIndex_myArrayType::myReductionEntry(NULL),thisProxy);
CProxySection_Hello::contribute(sizeof(int), &data, CkReduction::sum_int, cookie, cb);
```

As in an array reduction, users can use built-in reduction types (Section 2.2.3) or define his/her own reducer functions (Section 2.3.8).

Section Operations with Migrating Elements

When using a section reduction, you don't need to worry about migrations of array elements. When migration happens, an array element in the array section can still use the CkSectionInfo it stored previously for doing a reduction. Reduction messages will be correctly delivered but may not be as efficient until a new multicast spanning tree is rebuilt internally in the CkMulticast library. When a new spanning tree is rebuilt, an updated CkSectionInfo is passed along with a multicast message, so it is recommended that CkGetSectionInfo() function is always called when a multicast message arrives (as shown in the above SayHi example).

In the case where a multicast root migrates, the library must reconstruct the spanning tree to get optimal performance. One will get the following warning message if this is not done: "Warning: Multicast not optimized after multicast root migrated." In the current implementation, the user needs to initiate the rebuilding process using resetSection.

```
void Foo::pup(PUP::er & p) {
    // if I am multicast root and it is unpacking
    if (ismcastroot && p.isUnpacking()) {
        CProxySection_Foo fooProxy; // proxy for the section
        fooProxy.resetSection(fooProxy);

        // you may want to reset reduction client to root
        CkCallback *cb = new CkCallback(...);
    }
}
```

Cross Array Sections

Cross array sections contain elements from multiple arrays. Construction and use of cross array sections is similar to normal array sections with the following restrictions.

- Arrays in a section must all be of the same type.
- Each array must be enumerated by array ID.
- The elements within each array must be enumerated explicitly.

Note: cross section logic also works for groups with analogous characteristics.

Given three arrays declared thusly:

```
std::vector<CkArrayID> aidArr(3);
for (int i=0; i<3; i++) {
    CProxy_multisectiontest_arrayId Aproxy = CProxy_multisectiontest_
    ↪arrayId::ckNew(masterproxy.ckGetGroupID(), ArraySize);
    aidArr[i] = Aproxy.ckGetArrayID();
}
```

One can make a section including the lower half elements of all three arrays as follows:

```
int boundary = ArraySize/2;
int afloor = boundary;
int aceiling = ArraySize-1;
int asectionSize = aceiling-afloor+1;
// cross section lower half of each array
std::vector<std::vector<CkArrayIndex> > aelems(3);
for (int k=0; k<3; k++) {
    aelems[k].resize(asectionSize);
    for (int i=afloor, j=0; i<=aceiling; i++, j++)
        aelems[k][j] = CkArrayIndex1D(i);
}
CProxySection_multisectiontest_array1d arrayLowProxy(aidArr, aelems);
```

The resulting cross section proxy, as in the example `arrayLowProxy`, can then be used for multicasts in the same way as a normal array section.

Note: For simplicity the above example has all arrays and sections of uniform size. The size of each array and the number of elements in each array within a section can all be set independently. For a more concrete example on how to use cross array section reduction, please refer to: `examples/charm++/hello/xarraySection`.

Manual Delegation

By default Charm++ uses the `CkMulticast` library for optimized broadcasts and reductions on array sections, but advanced Charm++ users can choose to delegate¹³ sections to custom libraries (called delegation managers). Note that group sections are not automatically delegated to `CkMulticast` and hence must be manually delegated to this library to benefit from the optimized multicast tree implementation. This is explained here, and see `examples/charm++/groupsection` for an example.

While creating a chare array one can set the auto delegation flag to `false` in `CkArrayOptions` and the runtime system will not use the default `CkMulticast` library. A `CkMulticastMgr` (or any other delegation manager) group can then be created by the user, and any section delegated to it.

One only needs to create one delegation manager group, and it can serve all multicast/reduction delegations for different array/group sections in an application. In the following we show a manual delegation example using `CkMulticast` (the same can be applied to custom delegation managers):

```
CkArrayOptions opts(...);
opts.setSectionAutoDelegate(false); // manual delegation
CProxy_Hello arrayProxy = CProxy_Hello::ckNew(opts,...);
CProxySection_Hello sectProxy = CProxySection_Hello::ckNew(...);
CkGroupID mCastGrpId = CProxy_CkMulticastMgr::ckNew();
CkMulticastMgr *mCastGrp = CProxy_CkMulticastMgr(mCastGrpId).ckLocalBranch();

sectProxy.ckSectionDelegate(mCastGrp); // initialize section proxy

sectProxy.someEntry(...); // multicast via delegation library as before
```

One can also set the default branching factor when creating a `CkMulticastMgr` group. Sections created via this manager will use the specified branching factor for their multicast tree. For example:

```
CkGroupID mCastGrpId = CProxy_CkMulticastMgr::ckNew(3); // factor is 3
```

Contributing using a custom `CkMulticastMgr` group:

¹³ See chapter 2.4.3 for general information on message delegation in Charm++.

```
CkSectionInfo cookie;

void SayHi(HiMsg *msg)
{
    CkGetSectionInfo(cookie, msg); // update section cookie every time
    int data = thisIndex;
    CkCallback cb(CkIndex_myArrayType::myReductionEntry(NULL), thisProxy);
    mcastGrp->contribute(sizeof(int), &data, CkReduction::sum_int, cookie, cb);
}
```

Setting default reduction client for a section when using manual delegation:

```
CProxySection_Hello sectProxy;
CkMulticastMgr *mcastGrp = CProxy_CkMulticastMgr(mCastGrpId).ckLocalBranch();
mcastGrp->setReductionClient(sectProxy, new CkCallback(...));
```

Writing the pup method:

```
void Foo::pup(PUP::er & p) {
    // if I am multicast root and it is unpacking
    if (ismcastrout && p.isUnpacking()) {
        CProxySection_Foo fooProxy; // proxy for the section
        CkMulticastMgr *mg = CProxy_CkMulticastMgr(mCastGrpId).ckLocalBranch();
        mg->resetSection(fooProxy);

        // you may want to reset reduction client to root
        CkCallback *cb = new CkCallback(...);
        mg->setReductionClient(mcp, cb);
    }
}
```

2.3.6 Chare and Message Inheritance

Charm++ supports C++ like inheritance among Charm++ objects such as chares, groups, and messages, making it easier to keep applications modular and allowing reuse of code.

Chare Inheritance

Chare inheritance makes it possible to remotely invoke methods of a base chare from a proxy of a derived chare. Suppose a base chare is of type BaseChare, then the derived chare of type DerivedChare needs to be declared in the Charm++ interface file to be explicitly derived from BaseChare. Thus, the constructs in the .ci file should look like:

```
chare BaseChare {
    entry BaseChare(someMessage *);
    entry void baseMethod(void);
    ...
}
chare DerivedChare : BaseChare {
    entry DerivedChare(otherMessage *);
    entry void derivedMethod(void);
    ...
}
```

Note that the access specifier public is omitted, because Charm++ interface translator only needs to know about the public inheritance, and thus public is implicit. A Chare can inherit privately from other classes too, but the Charm++

interface translator does not need to know about it, because it generates support classes (*proxies*) to remotely invoke only public methods.

The class definitions of these chares should look like:

```
class BaseChare : public CBase_BaseChare {
    // private or protected data
public:
    BaseChare(someMessage *);
    void baseMethod(void);
};
class DerivedChare : public CBase_DerivedChare {
    // private or protected data
public:
    DerivedChare(otherMessage *);
    void derivedMethod(void);
};
```

It is possible to create a derived chare, and invoke methods of base chare from it, or to assign a derived chare proxy to a base chare proxy as shown below:

```
...
otherMessage *msg = new otherMessage();
CProxy_DerivedChare pd = CProxy_DerivedChare::ckNew(msg);
pd.baseMethod();      // OK
pd.derivedMethod();   // OK
...
CProxy_BaseChare pb = pd;
pb.baseMethod();      // OK
pb.derivedMethod();   // COMPILER ERROR
```

To pass constructor arguments from `DerivedChare::DerivedChare(someMessage*)` to `BaseChare::BaseChare(someMessage*)`, they can be forwarded through the `CBase` type constructor as follows:

```
DerivedChare::DerivedChare(someMessage *msg)
: CBase_DerivedChare(msg) // Will forward all arguments to BaseChare::BaseChare
{ }
```

If no arguments are provided, the generated C++ code for the `CBase_DerivedChare` constructor calls the default constructor of the base class `BaseChare`.

Entry methods are inherited in the same manner as methods of sequential C++ objects. To make an entry method virtual, just add the keyword `virtual` to the corresponding chare method declaration in the class header- no change is needed in the interface file. Pure virtual entry methods also require no special description in the interface file.

Inheritance for Messages

Messages cannot inherit from other messages. A message can, however, inherit from a regular C++ class. For example:

```
// In the .ci file:
message BaseMessage1;
message BaseMessage2;

// In the .h file:
class Base {
    // ...
};
```

(continues on next page)

(continued from previous page)

```

class BaseMessage1 : public Base, public CMessage_BaseMessage1 {
    // ...
};
class BaseMessage2 : public Base, public CMessage_BaseMessage2 {
    // ...
};

```

Messages cannot contain virtual methods or virtual base classes unless you use a packed message. Parameter marshalling has complete support for inheritance, virtual methods, and virtual base classes via the PUP::able framework.

2.3.7 Generic and Meta Programming with Templates

Templates are a mechanism provided by the C++ language to parametrize code over various types and constants with compile-time code specialization for each instance. Charm++ allows developers to implement various entities using C++ templates to gain their advantages in abstraction, flexibility, and performance. Because the Charm++ runtime system requires some generated code for each entity type that is used in a program, template entities must each have a declaration in a .ci file, a definition in a C++ header, and declarations of their instantiations in one or more .ci files.

The first step to implementing a templated Charm++ entity is declaring it as such in a .ci file. This declaration takes the same form as any C++ template: the `template` keyword, a list of template parameters surrounded by angle brackets, and the normal declaration of the entity with possible reference to the template parameters. The Charm++ interface translator will generate corresponding templated code for the entity, similar to what it would generate for a non-templated entity of the same kind. Differences in how one uses this generated code are described below.

A message template might be declared as follows:

```

module A {
    template <class DType, int N=3>
    message TMessage;
};

```

Note that default template parameters are supported.

If one wished to include variable-length arrays in a message template, those can be accommodated as well:

```

module B {
    template <class DType>
    message TVarMessage {
        DType payload[];
    };
};

```

Similarly, chare class templates (for various kinds of chares) would be written:

```

module C {
    template <typename T>
    chare TChare {
        entry TChare();
        entry void doStuff(T t);
    };

    template <typename U>
    group TGroup {
        entry TGroup();
        entry void doSomethingElse(U u, int n);
    };
};

```

(continues on next page)

(continued from previous page)

```
};

template <typename V, int s>
array [2D] TArray {
    entry TArray(V v);
};

template <typename W>
nodegroup TNodeGroup {
    entry TNodeGroup();
    entry void doAnotherThing(W w);
};
};
```

Entry method templates are declared like so:

```
module D {
    array [1D] libArray {
        entry libArray(int _dataSize);
        template <typename T>
        entry void doSomething(T t, CkCallback redCB);
    };
};
```

The definition of templated Charm++ entities works almost identically to the definition of non-template entities, with the addition of the expected template signature:

```
// A.h
#include "A.decl.h"

template <class DType, int N=3>
struct TMessage : public CMessage_TMessage<DType, N> {
    DType d[N];
};

#define CK_TEMPLATES_ONLY
#include "A.def.h"
#undef CK_TEMPLATES_ONLY
```

The distinguishing change is the additional requirement to include parts of the generated .def.h file that relate to the templates being defined. This exposes the generated code that provides registration and other supporting routines to client code that will need to instantiate it. As with C++ template code in general, the entire definition of the templated entity must be visible to the code that eventually references it to allow instantiation. In circumstances where module A contains only template code, some source file including A.def.h without the template macro will still have to be compiled and linked to incorporate module-level generated code.

Code that references particular templated entities needs to ask the interface translator to instantiate registration and delivery code for those entities. This is accomplished by a declaration in a .ci file that names the entity and the actual template arguments for which an instantiation is desired.

For the message and chare templates described above, a few instantiations might look like

```
module D {
    extern module A;
    message TMessage<float, 7>;
    message TMessage<double>;
```

(continues on next page)

(continued from previous page)

```

message TMessage<int, 1>;

extern module C;
array [2D] TArray<std::string, 4>;
group TGroup<char>;
};

```

Instantiations of entry method templates are slightly more complex, because they must specify the chare class containing them. The template arguments are also specified directly in the method's parameters, rather than as distinct template arguments.

```

module E {
    extern module D;

    // syntax: extern entry void chareClassName templateEntryMethodName(list, of, ↵
    ↵actual, arguments);
    extern entry void libArray doSomething(int&, CkCallback redCB);
};

```

To enable generic programming using Charm++ entities, we define a number of type trait utilities. These can be used to determine at compile-time if a type is a certain kind of Charm++ type:

```

#include "charm++_type_traits.h"

// Is T a chare array proxy?
using result = charmxx::is_array_proxy<T>;

// Is T a group proxy?
using result = charmxx::is_group_proxy<T>;

// Is T a node group proxy?
using result = charmxx::is_node_group_proxy<T>;

// Is T a chare proxy?
using result = charmxx::is_chare_proxy<T>;

// Is T a bound array?
using result = charmxx::is_bound_array<T>;

// Does T have a PUP routine defined for it?
using result = charmxx::is_pupable<T>;

```

2.3.8 Collectives

Reduction Clients

After the data is reduced, it is passed to you via a callback object, as described in section 2.3.2. The message passed to the callback is of type `CkReductionMsg`. Unlike typed reductions briefed in Section 2.2.3, here we discuss callbacks that take `CkReductionMsg*` argument. The important members of `CkReductionMsg` are `getSize()`, which returns the number of bytes of reduction data; and `getData()`, which returns a “void *” to the actual reduced data.

The callback to be invoked when the reduction is complete is specified as an additional parameter to contribute. It is an error for chare array elements to specify different callbacks to the same reduction contribution.

```
double forces[2]=get_my_forces();
// When done, broadcast the CkReductionMsg to "myReductionEntry"
CkCallback cb(CkIndex_myArrayType::myReductionEntry(NULL), thisProxy);
contribute(2*sizeof(double), forces,CkReduction::sum_double, cb);
```

In the case of the reduced version used for synchronization purposes, the callback parameter will be the only input parameter:

```
CkCallback cb(CkIndex_myArrayType::myReductionEntry(NULL), thisProxy);
contribute(cb);
```

and the corresponding callback function:

```
void myReductionEntry(CkReductionMsg *msg)
{
    int reducedArrSize=msg->getSize() / sizeof(double);
    double *output=(double *) msg->getData();
    for(int i=0 ; i<reducedArrSize ; i++)
    {
        // Do something with the reduction results in each output[i] array element
        .
        .
        .
    }
    delete msg;
}
```

(See examples/charm++/reductions/simple_reduction for a complete example).

If the target of a reduction is an entry method defined by a *when* clause in SDAG (Section 2.2.4), one may wish to set a reference number (or tag) that SDAG can use to match the resulting reduction message. To set the tag on a reduction message, call the `CkCallback::setRefNum(CMK_REFNUM_TYPE refnum)` method on the callback passed to the `contribute()` call.

Defining a New Reduction Type

It is possible to define a new type of reduction, performing a user-defined operation on user-defined data. This is done by creating a *reduction function*, which combines separate contributions into a single combined value.

The input to a reduction function is a list of `CkReductionMsgs`. A `CkReductionMsg` is a thin wrapper around a buffer of untyped data to be reduced. The output of a reduction function is a single `CkReductionMsg` containing the reduced data, which you should create using the `CkReductionMsg::buildNew(int nBytes,const void *data)` method.

Thus every reduction function has the prototype:

```
CkReductionMsg *reductionFn(int nMsg,CkReductionMsg **msgs);
```

For example, a reduction function to add up contributions consisting of two machine `short` ints would be:

```
CkReductionMsg *sumTwoShorts(int nMsg,CkReductionMsg **msgs)
{
    // Sum starts off at zero
    short ret[2]={0,0};
    for (int i=0;i<nMsg;i++) {
        // Sanity check:
        CkAssert(msgs[i]->getSize()==2*sizeof(short));
```

(continues on next page)

(continued from previous page)

```

// Extract this message's data
short *m=(short *)msgs[i]->getData();
ret[0]+=m[0];
ret[1]+=m[1];
}
return CkReductionMsg::buildNew(2*sizeof(short),ret);
}

```

The reduction function must be registered with Charm++ using `CkReduction::addReducer(reducerFn fn=NULL, bool streamable=false, const char* name=NULL)` from an `initnode` routine (see section 2.2.8 for details on the `initnode` mechanism). It takes a required parameter, `reducerFn fn`, a function pointer to the reduction function, and an optional parameter `bool streamable`, which indicates if the function is streamable or not (see section 2.3.8 for more information). `CkReduction::addReducer` returns a `CkReduction::reducerType` which you can later pass to `contribute`. Since `initnode` routines are executed once on every node, you can safely store the `CkReduction::reducerType` in a global or class-static variable. For the example above, the reduction function is registered and used in the following manner:

```

// In the .ci file:
initnode void registerSumTwoShorts(void);

// In some .C file:
/*global*/ CkReduction::reducerType sumTwoShortsType;
/*initnode*/ void registerSumTwoShorts(void)
{
    sumTwoShortsType=CkReduction::addReducer(sumTwoShorts);
}

// In some member function, contribute data to the customized reduction:
short data[2]=...;
contribute(2*sizeof(short),data,sumTwoShortsType);

```

Note that typed reductions briefed in Section 2.2.3 can also be used for custom reductions. The target reduction client can be declared as in Section 2.2.3 but the reduction functions will be defined as explained above.

Note that you cannot call `CkReduction::addReducer` from anywhere but an `initnode` routine.

(See `Reduction.cpp` of [Barnes-Hut MiniApp](#) for a complete example).

Streamable Reductions

For custom reductions over fixed sized messages, it is often desirable that the runtime process each contribution in a streaming fashion, i.e. as soon as a contribution is received from a chare array element, that data should be combined with the current aggregation of other contributions on that PE. This results in a smaller memory footprint because contributions are immediately combined as they come in rather than waiting for all contributions to be received. Users can write their own custom streamable reducers by reusing the message memory of the zeroth message in their reducer function by passing it as the last argument to `CkReduction::buildNew`:

```

CkReductionMsg *sumTwoShorts(int nMsg,CkReductionMsg **msgs)
{
    // reuse msgs[0]'s memory:
    short *retData = (short*)msgs[0]->getData();
    for (int i=1;i<nMsg;i++) {
        // Sanity check:
        CkAssert(msgs[i]->getSize()==2*sizeof(short));
    }
}

```

(continues on next page)

(continued from previous page)

```
// Extract this message's data
short *m=(short *)msgs[i]->getData();
retData[0]+=m[0];
retData[1]+=m[1];
}
return CkReductionMsg::buildNew(2*sizeof(short), retData, sumTwoShortsReducer,
↪msgs[0]);
}
```

Note that *only message zero* is allowed to be reused. For reducer functions that do not operate on fixed sized messages, such as set and concat, streaming would result in quadratic memory allocation and so is not desirable. Users can specify that a custom reducer is streamable when calling `CkReduction::addReducer` by specifying an optional boolean parameter (default is false). They can also provide a name string for their reducer to aid in debugging (default is NULL).

```
static void initNodeFn(void) {
    sumTwoShorts = CkReduction::addReducer(sumTwoShorts, /* streamable = */ true, /*
↪name = */ "sumTwoShorts");
}
```

2.3.9 Serializing Complex Types

This section describes advanced functionality in the PUP framework. The first subsections describe features supporting complex objects, with multiple levels of inheritance, or with dynamic changes in heap usage. The latter subsections describe additional language bindings, and features supporting PUP modes which can be used to copy object state from and to long-term storage for checkpointing, or other application level purposes.

Dynamic Allocation

If your class has fields that are dynamically allocated, when unpacking, these need to be allocated in the usual way before you pup them. Deallocation should be left to the class destructor as usual.

No allocation

The simplest case is when there is no dynamic allocation. Example:

```
class keepsFoo : public mySuperclass {
private:
    foo f; /* simple foo object */
public:
    keepsFoo(void) { }
    void pup(PUP::er &p) {
        mySuperclass::pup(p);
        p|f; // pup f's fields (calls f.pup(p));
    }
    ~keepsFoo() { }
};
```

Allocation outside pup

The next simplest case is when we contain a class that is always allocated during our constructor, and deallocated during our destructor. Then no allocation is needed within the pup routine.

```
class keepsHeapFoo : public mySuperclass {
private:
    foo *f; /* Heap-allocated foo object */
public:
    keepsHeapFoo(void) {
        f=new foo;
    }
    void pup(PUP::er &p) {
        mySuperclass::pup(p);
        p|*f; /* pup f's fields (calls f->pup(p))
    }
    ~keepsHeapFoo() { delete f; }
};
```

Allocation during pup

If we need values obtained during the pup routine before we can allocate the class, we must allocate the class inside the pup routine. Be sure to protect the allocation with `if (p.isUnpacking())`.

```
class keepsOneFoo : public mySuperclass {
private:
    foo *f; /* Heap-allocated foo object */
public:
    keepsOneFoo(...) { f=new foo(...); }
    keepsOneFoo() { f=NULL; } /* pup constructor */
    void pup(PUP::er &p) {
        mySuperclass::pup(p);
        // ...
        if (p.isUnpacking()) /* must allocate foo now */
            f=new foo(...);
        p|*f; /* pup f's fields
    }
    ~keepsOneFoo() { delete f; }
};
```

Allocatable array

For example, if we keep an array of doubles, we need to know how many doubles there are before we can allocate the array. Hence we must first pup the array length, do our allocation, and then pup the array data. We could allocate memory using malloc/free or other allocators in exactly the same way.

```
class keepsDoubles : public mySuperclass {
private:
    int n;
    double *arr; /* new'd array of n doubles */
public:
    keepsDoubles(int n_) {
        n=n_;
```

(continues on next page)

(continued from previous page)

```

    arr=new double[n];
}
keepsDoubles() { }

void pup(PUP::er &p) {
    mySuperclass::pup(p);
    p|n; // pup the array length n
    if (p.isUnpacking()) arr=new double[n];
    PUParray(p,arr,n); // pup data in the array
}

~keepsDoubles() { delete[] arr; }
};

```

NULL object pointer

If our allocated object may be NULL, our allocation becomes much more complicated. We must first check and pup a flag to indicate whether the object exists, then depending on the flag, pup the object.

```

class keepsNullFoo : public mySuperclass {
private:
    foo *f; /*Heap-allocated foo object, or NULL*/
public:
    keepsNullFoo(...) { if (...) f=new foo(...); }
    keepsNullFoo() { f=NULL; }
    void pup(PUP::er &p) {
        mySuperclass::pup(p);
        int has_f = (f!=NULL);
        p|has_f;
        if (has_f) {
            if (p.isUnpacking()) f=new foo;
            p|*f;
        } else {
            f=NULL;
        }
    }
    ~keepsNullFoo() { delete f; }
};

```

This sort of code is normally much longer and more error-prone if split into the various packing/unpacking cases.

Array of classes

An array of actual classes can be treated exactly the same way as an array of basic types. PUParray will pup each element of the array properly, calling the appropriate operator|.

```

class keepsFoos : public mySuperclass {
private:
    int n;
    foo *arr; /* new'd array of n foos */
public:
    keepsFoos(int n_) {
        n=n_;
    }
};

```

(continues on next page)

(continued from previous page)

```

    arr=new foo[n];
}
keepsFoos() { arr=NULL; }

void pup(PUP::er &p) {
    mySuperclass::pup(p);
    p|n; // pup the array length n
    if (p.isUnpacking()) arr=new foo[n];
    PUPArray(p,arr,n); // pup each foo in the array
}

~keepsFoos() { delete[] arr; }
};

```

Array of pointers to classes

An array of pointers to classes must handle each element separately, since the PUPArray routine does not work with pointers. An “allocate” routine to set up the array could simplify this code. More ambitious is to construct a “smart pointer” class that includes a pup routine.

```

class keepsFooPtrs : public mySuperclass {
private:
    int n;
    foo **arr; /* new'd array of n pointer-to-foos */
public:
    keepsFooPtrs(int n_) {
        n=n_;
        arr=new foo*[n]; // allocate array
        for (int i=0; i<n; i++) arr[i]=new foo(...); // allocate i'th foo
    }
    keepsFooPtrs() { arr=NULL; }

    void pup(PUP::er &p) {
        mySuperclass::pup(p);
        p|n; // pup the array length n
        if (p.isUnpacking()) arr=new foo*[n]; // allocate array
        for (int i=0; i<n; i++) {
            if (p.isUnpacking()) arr[i]=new foo(...); // allocate i'th foo
            p|*arr[i]; // pup the i'th foo
        }
    }

    ~keepsFooPtrs() {
        for (int i=0; i<n; i++) delete arr[i];
        delete[] arr;
    }
};

```

Note that this will not properly handle the case where some elements of the array are actually subclasses of foo, with virtual methods. The PUP::able framework described in the next section can be helpful in this case.

Subclass allocation via PUP::able

If the class *foo* above might have been a subclass, instead of simply using `new foo` above we would have had to allocate an object of the appropriate subclass. Since determining the proper subclass and calling the appropriate constructor yourself can be difficult, the PUP framework provides a scheme for automatically determining and dynamically allocating subobjects of the appropriate type.

Your superclass must inherit from `PUP::able`, which provides the basic machinery used to move the class. A concrete superclass and all its concrete subclasses require these four features:

- A line declaring `PUPable className;` in the `.ci` file. This registers the class' constructor. If `className` is a templated class, each concrete instantiation should have its own fully specified `PUPable` declaration in the `.ci` file.
- A call to the macro `PUPable_decl(className)` in the class' declaration, in the header file. This adds a virtual method to your class to allow `PUP::able` to determine your class' type. If `className` is a templated class, instead use `PUPable_decl_base_template(baseClassName, className)`, where `baseClassName` is the name of the base class. Both class names should include template specifications if necessary.
- A migration constructor — a constructor that takes `CkMigrateMessage *`. This is used to create the new object on the receive side, immediately before calling the new object's `pup` routine. Users should not free the `CkMigrateMessage`.
- A working, virtual `pup` method. You can omit this if your class has no data that needs to be packed.

As an added note for `PUP::able` classes which are templated: just as with templated chares, you will also need to include the `.def.h` surrounded by `CK_TEMPLATES_ONLY` preprocessor guards in an appropriate location, as described in Section 2.3.7.

An abstract superclass — a superclass that will never actually be packed — only needs to inherit from `PUP::able` and include a `PUPable_abstract(className)` macro in their body. For these abstract classes, the `.ci` file, `PUPable_decl` macro, and constructor are not needed.

For example, if *parent* is a concrete superclass, and *child* and *tchild* are its subclasses:

```
// ----- In the .ci file -----
PUPable parent;
PUPable child; // Could also have said `PUPable parent, child;`
// One PUPable declaration per concrete instantiation of tchild
PUPable tchild<int, double>;
PUPable tchild<char, Foo>;

// ----- In the .h file -----
class parent : public PUP::able {
    // ... data members ...
public:
    // ... other methods ...
    parent() {...}

    // PUP::able support: decl, migration constructor, and pup
    PUPable_decl(parent);
    parent(CkMigrateMessage *m) : PUP::able(m) {}
    virtual void pup(PUP::er &p) {
        PUP::able::pup(p); // Call base class
        // ... pup data members as usual ...
    }
};
```

(continues on next page)

(continued from previous page)

```

class child : public parent {
    // ... more data members ...
public:
    // ... more methods, possibly virtual ...
    child() {...}

    // PUP::able support: decl, migration constructor, and pup
    PUPable_decl(child);
    child(CkMigrateMessage *m) : parent(m) {}
    virtual void pup(PUP::er &p) {
        parent::pup(p); // Call base class
        // ... pup child's data members as usual ...
    }
};

template <typename T1, typename T2>
class tchild : public parent {
    // ... more data members ...
public:
    // ... more methods, possibly virtual ...
    tchild() { ... }

    // PUP::able support for templated classes
    // If parent were templated, we'd also include template args for parent
    PUPable_decl_base_template(parent, tchild<T1,T2>);
    tchild(CkMigrateMessage* m) : parent(m) {}
    virtual void pup(PUP::er &p) {
        parent::pup(p); // Call base class
        // ... pup tchild's data members as usual ...
    }
};

// Because tchild is a templated class with PUPable decls in the .ci file ...
#define CK_TEMPLATES_ONLY
#include "module_name.def.h"
#undef CK_TEMPLATES_ONLY

```

With these declarations, we can automatically allocate and pup a pointer to a parent or child using the vertical bar PUP::er syntax, which on the receive side will create a new object of the appropriate type:

```

class keepsParent {
    parent *obj; // May actually point to a child class (or be NULL)
public:
    // ...
    ~keepsParent() {
        delete obj;
    }
    void pup(PUP::er &p)
    {
        p|obj;
    }
};

```

This will properly pack, allocate, and unpack `obj` whether it is actually a parent or child object. The child class can use all the usual C++ features, such as virtual functions and extra private data.

If `obj` is NULL when packed, it will be restored to NULL when unpacked. For example, if the nodes of a binary tree

are PUP::able, one may write a recursive pup routine for the tree quite easily:

```
// ----- In the .ci file -----
PUPable treeNode;

// ----- In the .h file -----
class treeNode : public PUP::able {
    treeNode *left; // Left subtree
    treeNode *right; // Right subtree
    // ... other fields ...
public:
    treeNode(treeNode *l=NULL, treeNode *r=NULL);
    ~treeNode() { delete left; delete right; }

    // The usual PUP::able support:
    PUPable_decl(treeNode);
    treeNode(CkMigrateMessage *m) : PUP::able(m) { left = right = NULL; }
    void pup(PUP::er &p) {
        PUP::able::pup(p); // Call base class
        p|left;
        p|right;
        // ... pup other fields as usual ...
    }
};
```

This same implementation will also work properly even if the tree's internal nodes are actually subclasses of treeNode.

You may prefer to use the macros PUPable_def(className) and PUPable_reg(className) rather than using PUPable in the .ci file. PUPable_def provides routine definitions used by the PUP::able machinery, and should be included in exactly one source file at file scope. PUPable_reg registers this class with the runtime system, and should be executed exactly once per node during program startup.

Finally, a PUP::able superclass like *parent* above must normally be passed around via a pointer or reference, because the object might actually be some subclass like *child*. Because pointers and references cannot be passed across processors, for parameter marshalling you must use the special templated smart pointer classes CkPointer and CkReference, which only need to be listed in the .ci file.

A CkReference is a read-only reference to a PUP::able object — it is only valid for the duration of the method call. A CkPointer transfers ownership of the unmarshalled PUP::able to the method, so the pointer can be kept and the object used indefinitely.

For example, if the entry method bar needs a PUP::able parent object for in-call processing, you would use a CkReference like this:

```
// ----- In the .ci file -----
entry void barRef(int x, CkReference<parent> p);

// ----- In the .h file -----
void barRef(int x, parent &p) {
    // can use p here, but only during this method invocation
}
```

If the entry method needs to keep its parameter, use a CkPointer like this:

```
// ----- In the .ci file -----
entry void barPtr(int x, CkPointer<parent> p);
```

(continues on next page)

(continued from previous page)

```
// ----- In the .h file -----
void barPtr(int x, parent *p) {
    // can keep this pointer indefinitely, but must eventually delete it
}
```

Both `CkReference` and `CkPointer` are read-only from the send side — unlike messages, which are consumed when sent, the same object can be passed to several parameter marshalled entry methods. In the example above, we could do:

```
parent *p = new child;
someProxy.barRef(x, *p);
someProxy.barPtr(x, p); // Makes a copy of p
delete p; // We allocated p, so we destroy it.
```

C and Fortran bindings

C and Fortran programmers can use a limited subset of the `PUP::er` capability. The routines all take a handle named `pup_er`. The routines have the prototype:

```
void pup_type(pup_er p, type *val);
void pup_types(pup_er p, type *vals, int nVals);
```

The first call is for use with a single element; the second call is for use with an array. The supported types are `char`, `short`, `int`, `long`, `uchar`, `ushort`, `uint`, `ulong`, `float`, and `double`, which all have the usual C meanings.

A byte-packing routine

```
void pup_bytes(pup_er p, void *data, int nBytes);
```

is also provided, but its use is discouraged for cross-platform puping.

`pup_isSizing`, `pup_isPacking`, `pup_isUnpacking`, and `pup_isDeleting` calls are also available. Since C and Fortran have no destructors, you should actually deallocate all data when passed a deleting `pup_er`.

C and Fortran users cannot use `PUP::able` objects, seeking, or write custom `PUP::ers`. Using the C++ interface is recommended.

Common PUP::ers

The most common `PUP::ers` used are `PUP::sizer`, `PUP::toMem`, and `PUP::fromMem`. These are sizing, packing, and unpacking `PUP::ers`, respectively.

`PUP::sizer` simply sums up the sizes of the native binary representation of the objects it is passed. `PUP::toMem` copies the binary representation of the objects passed into a preallocated contiguous memory buffer. `PUP::fromMem` copies binary data from a contiguous memory buffer into the objects passed. All three support the `size` method, which returns the number of bytes used by the objects seen so far.

Other common `PUP::ers` are `PUP::toDisk`, `PUP::fromDisk`, and `PUP::xlater`. The first two are simple filesystem variants of the `PUP::toMem` and `PUP::fromMem` classes; `PUP::xlater` translates binary data from an unpacking `PUP::er` into the machine's native binary format, based on a `machineInfo` structure that describes the format used by the source machine.

An example of `PUP::toDisk` is available in `examples/charm++/PUP/pupDisk`.

PUP::seekBlock

It may rarely occur that you require items to be unpacked in a different order than they are packed. That is, you want a seek capability. *PUP::ers* support a limited form of seeking.

To begin a seek block, create a `PUP::seekBlock` object with your current `PUP::er` and the number of “sections” to create. Seek to a (0-based) section number with the `seek` method, and end the seeking with the `endBlock` method. For example, if we have two objects A and B, where A’s pup depends on and affects some object B, we can pup the two with:

```
void pupAB(PUP::er &p)
{
    // ... other fields ...
    PUP::seekBlock s(p,2); // 2 seek sections
    if (p.isUnpacking())
    { // In this case, pup B first
        s.seek(1);
        B.pup(p);
    }
    s.seek(0);
    A.pup(p,B);

    if (!p.isUnpacking())
    { // In this case, pup B last
        s.seek(1);
        B.pup(p);
    }
    s.endBlock(); // End of seeking block
    // ... other fields ...
};
```

Note that without the seek block, A’s fields would be unpacked over B’s memory, with *disastrous* consequences. The packing or sizing path must traverse the seek sections in numerical order; the unpack path may traverse them in any order. There is currently a small fixed limit of **3** on the maximum number of seek sections.

Writing a PUP::er

System-level programmers may occasionally find it useful to define their own `PUP::er` objects. The system `PUP::er` class is an abstract base class that funnels all incoming pup requests to a single subroutine:

```
virtual void bytes(void *p, int n, size_t itemSize, dataType t);
```

The parameters are, in order, the field address, the number of items, the size of each item, and the type of the items. The `PUP::er` is allowed to use these fields in any way. However, an `isSizing` or `isPacking` `PUP::er` may not modify the referenced user data; while an `isUnpacking` `PUP::er` may not read the original values of the user data. If your `PUP::er` is not clearly packing (saving values to some format) or unpacking (restoring values), declare it as sizing `PUP::er`.

2.3.10 Querying Network Topology

The following calls provide information about the machine upon which the parallel program is executed. A processing element (PE) is a unit of mapping and scheduling, which takes the form of an OS thread in SMP mode and an OS process in non-SMP mode. A node (specifically, a logical node) refers to an OS process: a set of one or more PEs that share memory (i.e. an address space). PEs and nodes are ranked separately starting from zero: PEs are ranked from 0 to `CmiNumPes()`, and nodes are ranked from 0 to `CmiNumNodes()`.

Charm++ provides a unified abstraction for querying topology information of IBM's BG/Q and Cray's XE6. The `TopoManager` singleton object, which can be used by including `TopoManager.h`, contains the following methods:

getDimNX(), getDimNY(), getDimNZ(): Returns the length of X, Y and Z dimensions (except BG/Q).

getDimNA(), getDimNB(), getDimNC(), getDimND(), getDimNE(): Returns the length of A, B, C, D and E dimensions on BG/Q.

getDimNT(): Returns the length of T dimension. `TopoManager` uses the T dimension to represent different cores that reside within a physical node.

rankToCoordinates(int pe, int &x, int &y, int &z, int &t): Get the coordinates of PE with rank *pe* (except BG/Q).

rankToCoordinates(int pe, int &a, int &b, int &c, int &d, int &e, int &t): Get the coordinates of PE with rank *pe* on BG/Q.

coordinatesToRank(int x, int y, int z, int t): Returns the rank of PE with given coordinates (except BG/Q).

coordinatesToRank(int a, int b, int c, int d, int e, int t): Returns the rank of PE with given coordinates on BG/Q.

getHopsBetweenRanks(int pe1, int pe2): Returns the distance between the given PEs in terms of the hops count on the network between the two PEs.

printAllocation(FILE *fp): Outputs the allocation for a particular execution to the given file.

For example, one can obtain the rank of a processor, whose coordinates are known, on Cray XE6 using the following code:

```
TopoManager *tmgr = TopoManager::getTopoManager();
int rank, x, y, z, t;
x = y = z = t = 2;
rank = tmgr->coordinatesToRank(x, y, z, t);
```

For more examples, please refer to `examples/charm++/topology`.

2.3.11 Physical Node API

The following calls provide information about the division and mapping of physical hardware in Charm++. A processing element (PE) is a unit of mapping and scheduling, which takes the form of an OS thread in SMP mode and an OS process in non-SMP mode. A logical node (often shortened to *node*) refers to an OS process: a set of one or more PEs that share memory (i.e. an address space). A physical node refers to an individual hardware machine (or, more precisely, an operating system instance on which Charm++ processes execute, or, in networking terminology, a *host*).

Communication between PEs on the same logical node is faster than communication between different logical nodes because OS threads share the same address space and can directly interact through shared memory. Communication between PEs on the same *physical* node may also be faster than between different physical nodes depending on the availability of OS features such as POSIX shared memory and Cross Memory Attach, the abilities of the network interconnect in use, and the speed of network loopback.

PEs are ranked in the range 0 to `CmiNumPes()`. Likewise, logical nodes are ranked from 0 to `CmiNumNodes()`, and physical nodes are ranked from 0 to `CmiNumPhysicalNodes()`.

Charm++ provides a set of functions for querying information about the mapping of PE's to physical nodes. The `cputopology.C` module, contains the following globally accessible functions:

int CmiPeOnSamePhysicalNode(int pe1, int pe2) Returns 1 if PEs *pe1* and *pe2* are on the same physical node and 0 otherwise.

int CmiNumPhysicalNodes() Returns the number of physical nodes that the program is running on.

int CmiNumPesOnPhysicalNode(int node) Returns the number of PEs that reside within a physical node.

void CmiGetPesOnPhysicalNode(int node, int **pelist, int *num) After execution `pelist` will point to a list of all PEs that reside within a physical `node` and `num` will point to the length of the list. One should be careful to not free or alter `pelist` since it points to reserved memory.

int CmiPhysicalRank(int pe) Returns the rank of a PE among all PEs running on the same physical node.

int CmiPhysicalNodeID(int pe) Returns the node ID of the physical node in which a PE resides.

int CmiGetFirstPeOnPhysicalNode(int node) Returns the lowest numbered processor on a physical node.

2.3.12 Checkpoint/Restart-Based Fault Tolerance

Charm++ offers two checkpoint/restart mechanisms. Each of these targets a specific need in parallel programming. However, both of them are based on the same infrastructure.

Traditional chare-array-based Charm++ applications, including AMPI applications, can be checkpointed to storage buffers (either files or memory regions) and be restarted later from those buffers. The basic idea behind this is straightforward: checkpointing an application is like migrating its parallel objects from the processors onto buffers, and restarting is the reverse. Thanks to the migration utilities like PUP methods (Section 2.2.5), users can decide what data to save in checkpoints and how to save them. However, unlike migration (where certain objects do not need a PUP method), checkpoint requires all the objects to implement the PUP method.

The two checkpoint/restart schemes implemented are:

- Shared filesystem: provides support for *split execution*, where the execution of an application is interrupted and later resumed.
- Double local-storage: offers an online *fault tolerance* mechanism for applications running on unreliable machines.

Split Execution

There are several reasons for having to split the execution of an application. These include protection against job failure, a single execution needing to run beyond a machine's job time limit, and resuming execution from an intermediate point with different parameters. All of these scenarios are supported by a mechanism to record execution state, and resume execution from it later.

Parallel machines are assembled from many complicated components, each of which can potentially fail and interrupt execution unexpectedly. Thus, parallel applications that take long enough to run from start to completion need to protect themselves from losing work and having to start over. They can achieve this by periodically taking a checkpoint of their execution state from which they can later resume.

Another use of checkpoint/restart is where the total execution time of the application exceeds the maximum allocation time for a job in a supercomputer. For that case, an application may checkpoint before the allocation time expires and then restart from the checkpoint in a subsequent allocation.

A third reason for having a split execution is when an application consists of *phases* and each phase may be run a different number of times with varying parameters. Consider, for instance, an application with two phases where the first phase only has a possible configuration (it is run only once). The second phase may have several configuration (for testing various algorithms). In that case, once the first phase is complete, the application checkpoints the result. Further executions of the second phase may just resume from that checkpoint.

An example of Charm++'s support for split execution can be seen in `tests/charm++/chkpt/hello`.

Checkpointing

The API to checkpoint the application is:

```
void CkStartCheckpoint(char* dirname, const CkCallback& cb, bool
requestStatus = false, int writersPerNode = 0);
```

The string `dirname` is the destination directory where the checkpoint files will be stored, and `cb` is the callback function which will be invoked after the checkpoint is done, as well as when the restart is complete. If `CkStartCheckpoint` is called again before `cb` has been called, the new request may be silently dropped. When the optional parameter `requestStatus` is true, the callback `cb` is sent a message of type `CkCheckpointStatusMsg` which includes an `int` `status` field of value `CK_CHECKPOINT_SUCCESS` or `CK_CHECKPOINT_FAILURE` indicating the success of the checkpointing operation. `writersPerNode` is an optional parameter that controls the number of PEs per logical node simultaneously allowed to write checkpoints. By default, it allows all PEs on a node to write at once, but should be tuned for large runs to avoid overloading the filesystem. Once set, this value persists for future calls to `CkStartCheckpoint`, so it does not need to be provided on every invocation (specifying 0 also leaves it at its current value).

Here is an example of a typical use:

```
/* ... */ CkCallback cb(CkIndex_Hello::SayHi(), helloProxy);
CkStartCheckpoint("log", cb);
```

A chare array usually has a PUP routine for the sake of migration. The PUP routine is also used in the checkpointing and restarting process. Therefore, it is up to the programmer what to save and restore for the application. One illustration of this flexibility is a complicated scientific computation application with 9 matrices, 8 of which hold intermediate results and 1 that holds the final results of each timestep. To save resources, the PUP routine can well omit the 8 intermediate matrices and checkpoint the matrix with the final results of each timestep.

Group, nodegroup (Section 2.2.7), and singleton chare objects are normally not meant to be migrated. In order to checkpoint them, however, the user has to write PUP routines for the groups and chare and declare them as `[migratable]` in the `.ci` file. Some programs use `mainchares` to hold key control data like global object counts, and thus `mainchares` need to be checkpointed too. To do this, the programmer should write a PUP routine for the `mainchares` and declare them as `[migratable]` in the `.ci` file, just as in the case of group and nodegroup.

Variables marked as `readonly` are automatically checkpointed and restored by the runtime system, so user PUP routines do not need to explicitly handle them.

Checkpointing variables of type `CProxy_*` is completely valid. After the restart callback is invoked, the original chare structures will have been reconstructed and all proxy variables will be valid references to the restarted versions of whatever they originally referred to. The only caveat to this is when the application is restarted on a different number of processors than it was checkpointed on, in which case non-location invariant chares have special behavior: singleton chares are not created or restored at all and group/nodegroup chares are created per PE/node, but each group/nodegroup element is restored from the checkpoint corresponding to the element originally on PE/node 0. This does not effect `mainchares`, which are always restarted on PE 0.

The checkpoint must be recorded at a synchronization point in the application, to ensure a consistent state upon restart. One easy way to achieve this is to synchronize through a reduction to a single chare (such as the `mainchare` used at startup) and have that chare make the call to initiate the checkpoint.

After `CkStartCheckpoint` is executed, a directory of the designated name is created and a collection of checkpoint files are written into it.

Note: Note that checkpoints are written to and read from several automatically created subdirectories of the specified directory in order to avoid creating too many files in the same directory, which can stress the file system.

Restarting

The user can choose to run the Charm++ application in restart mode, i.e., restarting execution from a previously-created checkpoint. The command line option `+restart DIRNAME` is required to invoke this mode. For example:

```
$ ./charmrun hello +p4 +restart log
```

Restarting is the reverse process of checkpointing. Charm++ allows restarting the old checkpoint on a different number of physical processors. This provides the flexibility to expand or shrink your application when the availability of computing resources changes.

When restarting, the runtime system recreates the state of the application based on the recorded log files, first restoring read-only variables, then mainchares, then singleton chares (only when the number of PEs in the restart matches the original number of PEs), then groups, then nodegroups, then chare arrays. Dependencies in group creation are implicitly respected since groups are recreated in the same order as they were originally created in on every PE (the same holds for nodegroups). Finally, when the state has been restored, the callback specified when the checkpoint was created is executed, restarting the application.

Note that on restart, if an array or group reduction client was set to a static function, the function pointer might be lost and the user needs to register it again. A better alternative is to always use an entry method of a chare object. Since all the entry methods are registered inside Charm++ system, in the restart phase, the reduction client will be automatically restored.

After a failure, the system may contain fewer or more processors. Once the failed components have been repaired, some processors may become available again. Therefore, the user may need the flexibility to restart on a different number of processors than in the checkpointing phase. This is allowable by giving a different `+pN` option at runtime. One thing to note is that the new load distribution might differ from the previous one at checkpoint time, so running a load balancer (see Section 2.2.6) after restart is suggested.

If restart is not done on the same number of processors, the processor-specific data in a group/nodegroup branch cannot (and usually should not) be restored individually. A copy from processor 0 will be propagated to all the processors. Additionally, singleton chares will not be restored at all in this case, so they must be specially handled when restarting on a different number of processors. One can, for example, explicitly test for a new number of PEs when unpacking and reconstruct singleton objects in that case.

Choosing What to Save

In your programs, you may use chare groups for different types of purposes. For example, groups holding read-only data can avoid excessive data copying, while groups maintaining processor-specific information are used as a local manager of the processor. In the latter situation, the data is sometimes too complicated to save and restore but easy to re-compute. For the read-only data, you want to save and restore it in the PUP'er routine and leave empty the migration constructor, via which the new object is created during restart. For the easy-to-recompute type of data, we just omit the PUP'er routine and do the data reconstruction in the group's migration constructor.

A similar example is the program mentioned above, where there are two types of chare arrays, one maintaining intermediate results while the other type holds the final result for each timestep. The programmer can take advantage of the flexibility by leaving PUP'er routine empty for intermediate objects, and do save/restore only for the important objects.

Online Fault Tolerance

As supercomputers grow in size, their reliability decreases correspondingly. This is due to the fact that the ability to assemble components in a machine surpasses the increase in reliability per component. What we can expect in the future is that applications will run on unreliable hardware.

The previous disk-based checkpoint/restart can be used as a fault tolerance scheme. However, it would be a very basic scheme in that when a failure occurs, the whole program gets killed and the user has to manually restart the application from the checkpoint files. The double local-storage checkpoint/restart protocol described in this subsection provides an automatic fault tolerance solution. When a failure occurs, the program can automatically detect the failure and restart from the checkpoint. Further, this fault-tolerance protocol does not rely on any reliable external storage (as needed in the previous method). Instead, it stores two copies of checkpoint data to two different locations (can be memory or local disk). This double checkpointing ensures the availability of one checkpoint in case the other is lost. The double in-memory checkpoint/restart scheme is useful and efficient for applications with small memory footprint at the checkpoint state. The double in-disk variant stores checkpoints into local disk, thus can be useful for applications with large memory footprint.

Checkpointing

The function that application developers can call to record a checkpoint in a chare-array-based application is:

```
void CkStartMemCheckpoint (CkCallback &cb)
```

where `cb` has the same meaning as in section 2.3.12. Just like the above disk checkpoint described, it is up to the programmer to decide what to save. The programmer is responsible for choosing when to activate checkpointing so that the size of a global checkpoint state, and consequently the time to record it, is minimized.

In AMPI applications, the user just needs to create an `MPI_Info` object with the key `"ampi_checkpoint"` and a value of either `"in_memory"` (for a double in-memory checkpoint) or `"to_file=file_name"` (to checkpoint to disk), and pass that object to the function `AMPI_Migrate()` as in the following:

```
// Setup
MPI_Info in_memory, to_file;

MPI_Info_create(&in_memory);
MPI_Info_set(in_memory, "ampi_checkpoint", "in_memory");

MPI_Info_create(&to_file);
MPI_Info_set(to_file, "ampi_checkpoint", "to_file=chkpt_dir");

...

// Main time-stepping loop
for (int iter=0; iter < max_iters; iter++) {

    // Time step work ...

    if (iter % chkpt_freq == 0)
        AMPI_Migrate(in_memory);
}
```

Restarting

When a processor crashes, the restart protocol will be automatically invoked to recover all objects using the last checkpoints. The program will continue to run on the surviving processors. This is based on the assumption that there are no extra processors to replace the crashed ones.

However, if there are a pool of extra processors to replace the crashed ones, the fault-tolerance protocol can also take advantage of this to grab one free processor and let the program run on the same number of processors as before the crash. In order to achieve this, Charm++ needs to be compiled with the macro option `CK_NO_PROC_POOL` turned on.

Double in-disk checkpoint/restart

A variant of double memory checkpoint/restart, *double in-disk checkpoint/restart*, can be applied to applications with large memory footprint. In this scheme, instead of storing checkpoints in the memory, it stores them in the local disk. The checkpoint files are named `ckpt [CkMyPe] - [idx] -XXXXX` and are stored under the `/tmp` directory.

Users can pass the runtime option `+ftc_disk` to activate this mode. For example:

```
./charmrun hello +p8 +ftc_disk
```

Building Instructions

In order to have the double local-storage checkpoint/restart functionality available, the parameter `syncft` must be provided at build time:

```
./build charm++ netlrts-linux-x86_64 syncft
```

At present, only a few of the machine layers underlying the Charm++ runtime system support resilient execution. These include the TCP-based `net` builds on Linux and Mac OS X. For clusters overbearing job-schedulers that kill a job if a node goes down, the way to demonstrate the killing of a process is show in Section 2.3.12. Charm++ runtime system can automatically detect failures and restart from checkpoint.

Failure Injection

To test that your application is able to successfully recover from failures using the double local-storage mechanism, we provide a failure injection mechanism that lets you specify which PEs will fail at what point in time. You must create a text file with two columns. The first column will store the PEs that will fail. The second column will store the time at which the corresponding PE will fail. Make sure all the failures occur after the first checkpoint. The runtime parameter `kill_file` has to be added to the command line along with the file name:

```
$ ./charmrun hello +p8 +kill_file <file>
```

An example of this usage can be found in the `syncfttest` targets in `tests/charm++/jacobi3d`.

Failure Demonstration

For HPC clusters, the job-schedulers usually kills a job if a node goes down. To demonstrate restarting after failures on such clusters, `CkDieNow()` function can be used. You just need to place it at any place in the code. When it is called by a processor, the processor will hang and stop responding to any communication. A spare processor will replace the crashed processor and continue execution after getting the checkpoint of the crashed processor. To make it work, you need to add the command line option `+wp`, the number following that option is the working processors and the remaining are the spare processors in the system.

2.3.13 Support for Loop-level Parallelism

To better utilize the multicore chip, it has become increasingly popular to adopt shared-memory multithreading programming methods to exploit parallelism on a node. For example, in hybrid MPI programs, OpenMP is the most popular choice. When launching such hybrid programs, users have to make sure there are spare physical cores allocated to the shared-memory multithreading runtime. Otherwise, the runtime that handles distributed-memory programming may interfere with resource contention because the two independent runtime systems are not coordinated. If spare cores are allocated, in the same way of launching a MPI+OpenMP hybrid program, Charm++ will work perfectly

with any shared-memory parallel programming languages (e.g. OpenMP). As with ordinary OpenMP applications, the number of threads used in the OpenMP parts of the program can be controlled with the `OMP_NUM_THREADS` environment variable. See Section 2.6.3 for details on how to propagate such environment variables.

If there are no spare cores allocated, to avoid resource contention, a *unified runtime* is needed to support both intra-node shared-memory multithreading parallelism and inter-node distributed-memory message-passing parallelism. Additionally, considering that a parallel application may have only a small fraction of its critical computation be suitable for porting to shared-memory parallelism (the savings on critical computation may also reduce the communication cost, thus leading to more performance improvement), dedicating physical cores on every node to the shared-memory multithreading runtime will waste computational power because those dedicated cores are not utilized at all during most of the application's execution time. This case indicates the necessity of a unified runtime supporting both types of parallelism.

CkLoop library

The *CkLoop* library is an add-on to the Charm++ runtime to achieve such a unified runtime. The library implements a simple OpenMP-like shared-memory multithreading runtime that reuses Charm++ PEs to perform tasks spawned by the multithreading runtime. This library targets the SMP mode of Charm++.

The *CkLoop* library is built in `$CHARM_DIR/$MACH_LAYER/tmp/libs/ck-libs/ckloop` by executing `make`. To use it for user applications, one has to include `CkLoopAPI.h` in the source code. The interface functions of this library are as follows:

- `CProxy_FuncCkLoop CkLoop_Init(int numThreads=0)`: This function initializes the CkLoop library, and it only needs to be called once on a single PE during the initialization phase of the application. The argument `numThreads` is only used in non-SMP mode, specifying the number of threads to be created for the single-node shared-memory parallelism. It will be ignored in SMP mode.
- `void CkLoop_SetSchedPolicy(CkLoop_sched schedPolicy=CKLOOP_NODE_QUEUE)`: This function sets the scheduling policy of CkLoop work, three options available: `CKLOOP_NODE_QUEUE`, `CKLOOP_TREE` and `CKLOOP_LIST`. The default policy, `CKLOOP_NODE_QUEUE` on supported environments is to use `node_queue` message so that master or another idle PE delivers the CkLoop work to all other PEs. `CKLOOP_TREE` policy is set by default for builds not supporting a node queue. This policy delivers CkLoop messages on the implicit tree. `CKLOOP_LIST` uses list to deliver the messages.
- `void CkLoop_Exit(CProxy_FuncCkLoop ckLoop)`: This function is intended to be used in non-SMP mode, as it frees the resources (e.g. terminating the spawned threads) used by the CkLoop library. It should be called on just one PE.
- `void CkLoop_Parallelize(HelperFn func, /* the function that finishes partial work on another thread */
int paramNum, /* the number of parameters for func */
void * param, /* the input parameters for the above func */
int numChunks, /* number of chunks to be partitioned */
int lowerRange, /* lower range of the loop-like parallelization [lowerRange, upperRange] */
int upperRange, /* upper range of the loop-like parallelization [lowerRange, upperRange] */
int sync=1, /* toggle implicit barrier after each parallelized loop */
void *redResult=NULL, /* the reduction result, ONLY SUPPORT SINGLE VAR of TYPE int/float/double */
REDUCTION_TYPE type=CKLOOP_NONE /* type of the reduction result */
CallerFn cfunc=NULL, /* caller PE will call this function before ckloop is done and before starting to work on its chunks */
int cparamNum=0, void *cparam=NULL /* the input parameters to the above function */
)`

The “HelperFn” is defined as “typedef void (*HelperFn)(int first,int last, void *result, int paramNum, void *param);” and the “result” is the buffer for reduction result on a single simple-type variable. The “CallerFn” is defined as “typedef void (*CallerFn)(int paramNum, void *param);”

Lambda syntax for *CkLoop* is also supported. The interface for using lambda syntax is as follows:

```
void CkLoop_Parallelize(
int numChunks, int lowerRange, int upperRange,
    [=](int first, int last, void *result) {
    for (int i = first; i <= last; ++i) {
        // work to parallelize goes here
    }
}, void *redResult=NULL, REDUCTION_TYPE type=CKLOOP_NONE,
std::function<void()> cfunc=NULL
);
```

Examples using this library can be found in `examples/charm++/ckloop` and the widely used molecular dynamics simulation application `NAMD`¹⁴.

The CkLoop Hybrid library

The `CkLoop_Hybrid` library is a mode of `CkLoop` that incorporates specific adaptive scheduling strategies aimed at providing a tradeoff between dynamic load balance and spatial locality. It is used in a build of Charm++ where all chares are placed on core 0 of each node (called the drone-mode, or all-drones-mode). It incorporates a strategy called staggered static-dynamic scheduling (from dissertation work of Vivek Kale). The iteration space is first tentatively divided approximately equally to all available PEs. Each PE’s share of the iteration space is divided into a static portion, specified by the `staticFraction` parameter below, and the remaining dynamic portion. The dynamic portion of a PE is divided into chunks of specified chunksize, and enqueued in the task-queue associated with that PE. Each PE works on its static portion, and then on its own task queue (thus preserving spatial locality, as well as persistence of allocations across outer iterations), and after finishing that, steals work from other PE’s task queues.

`CkLoopHybrid` support requires the SMP mode of Charm++ and the additional flags `-enable-drone-mode` and `-enable-task-queue` to be passed as build options when Charm++ is built.

The changes to the `CkLoop` API call are the following:

- `CkLoop_Init` does not need to be called
- `CkLoop_SetSchedPolicy` is not supported
- `CkLoop_Exit` does not need to be called
- `CkLoop_Parallelize` call is similar to `CkLoop` but has an additional variable that provides the fraction of iterations that are statically scheduled:

```
void CkLoop_ParallelizeHybrid(
float staticFraction,
HelperFn func, /* the function that finishes partial work on another thread */
int paramNum, /* the number of parameters for func */
void * param, /* the input parameters for the above func */
int numChunks, /* number of chunks to be partitioned */
int lowerRange, /* lower range of the loop-like parallelization [lowerRange, ↵
↵upperRange] */
int upperRange, /* upper range of the loop-like parallelization [lowerRange, ↵
↵upperRange] */
);
```

(continues on next page)

¹⁴ <http://www.ks.uiuc.edu/Research/namd>

(continued from previous page)

```

int sync=1, /* toggle implicit barrier after each parallelized loop */
void *redResult=NULL, /* the reduction result, ONLY SUPPORT SINGLE VAR of TYPE_
↳int/float/double */
REDUCTION_TYPE type=CKLOOP_NONE /* type of the reduction result */
CallerFn cfunc=NULL, /* caller PE will call this function before ckloop is done_
↳and before starting to work on its chunks */
int cparamNum=0, void *cparam=NULL /* the input parameters to the above function_
↳ */
)

```

Reduction is supported for type CKLOOP_INT_SUM, CKLOOP_FLOAT_SUM, CKLOOP_DOUBLE_SUM. It is recommended to use this mode without reduction.

Charm++/Converse Runtime Scheduler Integrated OpenMP

The compiler-provided OpenMP runtime library can work with Charm++ but it creates its own thread pool so that Charm++ and OpenMP can have oversubscription problem. The integrated OpenMP runtime library parallelizes OpenMP regions in each chare and runs on the Charm++ runtime without oversubscription. The integrated runtime creates OpenMP user-level threads, which can migrate among PEs within a node. This fine-grained parallelism by the integrated runtime helps resolve load imbalance within a node easily. When PEs become idle, they help other busy PEs within a node via work-stealing.

Instructions to build and use the integrated OpenMP library

Instructions to build

The OpenMP library can be built with `omp` keyword on any `smp` version of Charm++ including multicore build when you build Charm++ or AMPI, for example:

```

$ $CHARM_DIR/build charm++ multicore-linux-x86_64 omp
$ $CHARM_DIR/build charm++ netlrts-linux-x86_64 smp omp

```

This library is based on the LLVM OpenMP runtime library. So it supports the ABI used by clang, intel and gcc compilers. The following is the list of compilers which are verified to support this integrated library on Linux.

- GCC: 4.8 or newer
- ICC: 15.0 or newer
- Clang: 3.7 or newer

You can use this integrated OpenMP with `clang` on IBM Blue Gene machines without special compilation flags (don't need to add `-fopenmp` or `-openmp` on Blue Gene clang).

On Linux, the OpenMP supported version of clang has been installed in default recently. For example, Ubuntu has been released with clang higher than 3.7 since 15.10. Depending on which version of clang is installed in your working environments, you should follow additional instructions to use this integrated OpenMP with Clang. The following is the instruction to use clang on Ubuntu where the default clang is older than 3.7. If you want to use clang on other Linux distributions, you can use package managers on those Linux distributions to install clang and OpenMP library. This installation of clang will add headers for OpenMP environmental routines and allow you to parse the OpenMP directives. However, on Ubuntu, the installation of clang doesn't come with its OpenMP runtime library so it results in an error message saying that it fails to link the compiler provided OpenMP library. This library is not needed to use the integrated OpenMP runtime but you need to avoid this error to succeed compiling your codes. The following are the instructions to avoid the error:


```
# When you want to compile Integrated OpenMP on Ubuntu where the pre-installed clang
# is older than 3.7, you can use integrated openmp with the following instructions.
# e.g.) Ubuntu 14.04, the version of default clang is 3.4.
$ sudo apt-get install clang-3.8 //you can use any version of clang higher than 3.8
$ sudo ln -svT /usr/bin/clang-3.8 /usr/bin/clang
$ sudo ln -svT /usr/bin/clang++-3.8 /usr/bin/clang

$ $CHARM_DIR/build charm++ multicore-linux-x86_64 clang omp --with-production -j8
$ echo '!<arch>' > $(CHARM_DIR)/lib/libomp.a # Dummy library. This will make you_
→avoid the error message.
```

On Mac, the Apple-provided clang installed in default doesn't have OpenMP feature. We're working on the support of this library on Mac with OpenMP enabled clang which can be downloaded and installed through Homebrew or MacPorts. Currently, this integrated library is built and compiled on Mac with the normal GCC which can be downloaded and installed via Homebrew and MacPorts. If installed globally, GCC will be accessible by appending the major version number and adding it to the invocation of the Charm++ build script. For example:

```
$ $CHARM_DIR/build charm++ multicore-linux-x86_64 omp gcc-7
$ $CHARM_DIR/build charm++ netlrts-linux-x86_64 smp omp gcc-7
```

If this does not work, you should set environment variables so that the Charm++ build script uses the normal gcc installed from Homebrew or MacPorts. The following is an example using Homebrew on Mac OS X 10.12.5:

```
# Install Homebrew from https://brew.sh
# Install gcc using 'brew' */
$ brew install gcc

# gcc, g++ and other binaries are installed at /usr/local/Cellar/gcc/<version>/bin
# You need to make symbolic links to the gcc binaries at /usr/local/bin
# In this example, gcc 7.1.0 is installed at the directory.
$ cd /usr/local/bin
$ ln -sv /usr/local/Cellar/gcc/7.1.0/bin/gcc-7 gcc
$ ln -sv /usr/local/Cellar/gcc/7.1.0/bin/g++-7 g++
$ ln -sv /usr/local/Cellar/gcc/7.1.0/bin/gcc-nm-7 gcc-nm
$ ln -sv /usr/local/Cellar/gcc/7.1.0/bin/gcc-ranlib-7 gcc-ranlib
$ ln -sv /usr/local/Cellar/gcc/7.1.0/bin/gcc-ar-7 gcc-ar

# Finally, you should set PATH variable so that these binaries are accessed first in_
→the build script.
$ export PATH=/usr/local/bin:$PATH
```

In addition, this library will be supported on Windows in the next release of Charm++.

How to use the integrated OpenMP on Charm++

To use this library on your applications, you have to add `-module OmpCharm` in compile flags to link this library instead of the compiler-provided library in compilers. Without `-module OmpCharm`, your application will use the compiler-provided OpenMP library which running on its own separate runtime (you don't need to add `-fopenmp` or `-openmp` with gcc and icc. These flags are included in the predefined compile options when you build Charm++ with omp).

This integrated OpenMP adjusts the number of OpenMP instances on each chare so the number of OpenMP instances can be changed for each OpenMP region over execution. If your code shares some data structures among OpenMP instances in a parallel region, you can set the size of the data structures before the start of the OpenMP region with `omp_get_max_threads()` and use the data structure within each OpenMP instance with

`omp_get_thread_num()`. After the OpenMP region, you can iterate over the data structure to combine partial results with `CmiGetCurKnownOmpThreads()`. `CmiGetCurKnownOmpThreads()` returns the number of OpenMP threads for the latest OpenMP region on the PE where a chare is running. The following is an example to describe how you can use shared data structures for OpenMP regions on the integrated OpenMP with Charm++:

```
/* Maximum possible number of OpenMP threads in the upcoming OpenMP region.
   Users can restrict this number with 'omp_set_num_threads()' for each chare
   and the environmental variable, 'OMP_NUM_THREADS' for all chares.
   By default, omp_get_max_threads() returns the number of PEs for each logical node.
*/
int maxAvailableThreads = omp_get_max_threads();
int *partialResult = new int[maxAvailableThreads]{0};

/* Partial sum for subsets of iterations assigned to each OpenMP thread.
   The integrated OpenMP runtime decides how many OpenMP threads to create
   with some heuristics internally.
*/
#pragma omp parallel for
for (int i = 0; i < 128; i++) {
    partialResult[omp_get_thread_num()] += i;
}

/* We can know how many OpenMP threads are created in the latest OpenMP region
   by CmiCurKnownOmpthreads().
   You can get partial results each OpenMP thread generated */
for (int j = 0; j < CmiCurKnownOmpThreads(); j++)
    CkPrintf("partial sum of thread %d: %d \n", j, partialResult[j]);
```

The list of supported pragmas

This library is forked from LLVM OpenMP Library supporting OpenMP 4.0. Among many number of directives specified in OpenMP 4.0, limited set of directives are supported. The following list of supported pragmas is verified from the OpenMP conformance test suite which forked from LLVM OpenMP library and ported to Charm++ program running multiple OpenMP instances on chares. The test suite can be found in `tests/converse/openmp_test`.

```
/* omp_<directive>_<clauses> */
omp_atomic
omp_barrier
omp_critical
omp_flush

/* the following directives means they work within omp parallel region */
omp_for_firstprivate
omp_for_lastprivate
omp_for_nowait
omp_for_private
omp_for_reduction
omp_for_schedule_dynamic
omp_for_schedule_guided
omp_for_schedule_static
omp_section_firstprivate
omp_section_lastprivate
omp_section_private
omp_sections_nowait
omp_sections_reduction

omp_get_num_threads
```

(continues on next page)

(continued from previous page)

```

omp_get_wtick
omp_get_wtime
omp_in_parallel
omp_master
omp_master_3
omp_parallel_default
omp_parallel_firstprivate

/* the following directives means the combination of 'omp parallel and omp for/section
↪ ' works */
omp_parallel_for_firstprivate
omp_parallel_for_if
omp_parallel_for_lastprivate
omp_parallel_for_private
omp_parallel_for_reduction
omp_parallel_sections_firstprivate
omp_parallel_sections_lastprivate
omp_parallel_sections_private
omp_parallel_sections_reduction

omp_parallel_if
omp_parallel_private
omp_parallel_reduction
omp_parallel_shared
omp_single
omp_single_nowait
omp_single_private

```

The other directives in OpenMP standard will be supported in the next version.

A simple example using this library can be found in `examples/charm++/openmp`. You can compare CkLoop and the integrated OpenMP with this example. You can see that the total execution time of this example with enough big size of problem is faster with OpenMP than CkLoop thanks to load balancing through work-stealing between threads within a node while the execution time of each chare can be slower on OpenMP because idle PEs helping busy PEs.

API to control which PEs participating in CkLoop/OpenMP work

User may want certain PE not to be involved in other PE's loop-level parallelization for some cases because it may add latency to works in the PE by helping other PEs. User can enable or disable each PE to participate in the loop-level parallelization through the following API:

void **CkSetPeHelpsOtherThreads** (int value)

value can be 0 or 1, 0 means this API disable the current PE to help other PEs. value 1 or others can enable the current PE for loop-level parallelization. By default, all the PEs are enabled for the loop-level parallelization by CkLoop and OpenMP. User should explicitly enable the PE again by calling this API with value 1 after they disable it during certain procedure so that the PE can help others after that. The following example shows how this API can be used.

```

CkSetPeHelpsOtherThreads(0);

/* codes which can run without the current PE
interrupted by loop-level work from other PEs */

CkSetPeHelpsOtherThreads(1);

```

2.3.14 GPU Support

Overview

GPUs are throughput-oriented devices with peak computational capabilities that greatly surpass equivalent-generation CPUs but with limited control logic. This currently constrains them to be used as accelerator devices controlled by code on the CPU. Traditionally, programmers have had to either (a) halt the execution of work on the CPU whenever issuing GPU work to simplify synchronization or (b) issue GPU work asynchronously and carefully manage and synchronize concurrent GPU work in order to ensure progress and good performance. The latter becomes significantly more difficult with overdecomposition as in Charm++, where numerous concurrent objects launch computational kernels and initiate data transfers on the GPU.

The support for GPUs in the Charm++ runtime system consist of the **GPU Manager** module and **Hybrid API (HAPI)**. HAPI exposes the core functionalities of GPU Manager to the Charm++ user. Currently only NVIDIA GPUs (and CUDA) are supported, although we are actively working on providing support for AMD and Intel GPUs as well. CUDA code can be integrated in Charm++ just like any C/C++ program to offload computational kernels, but when used naively, performance will most likely be far from ideal. This is because overdecomposition, a core concept of Charm++, creates relatively fine-grained objects and tasks that needs support from the runtime to be executed efficiently.

We strongly recommend using CUDA Streams to assign one more streams to each chare, so that multiple GPU operations initiated by different chares can be executed concurrently when possible. Concurrent Kernel Execution in CUDA allows kernels in different streams to execute concurrently on the device unless a single kernel uses all of the available GPU resources. It should be noted that concurrent data transfers (even when they are in different streams) are limited by the number of DMA engines on the GPU, and most current GPUs have only one engine per direction (host-to-device, device-to-host).

In addition to using CUDA Streams to maximize concurrency, another important consideration is avoiding synchronization calls such as `cudaStreamSynchronize` or `cudaDeviceSynchronize`. This is because the chare that has just enqueued some work to the GPU should yield the PE, in order to allow other chares waiting on the same PE to execute. Because of the message-driven execution pattern in Charm++, it is infeasible for the user to asynchronously detect when a GPU operation completes, either by manually polling the GPU or adding a CUDA Callback to the stream. One of the core functionalities of GPU Manager is supporting this asynchronous detection, which allows a Charm++ callback to be invoked when all previous operations in a specified CUDA stream complete. This is exposed to the user via a HAPI call, which is demonstrated in the usage section below.

Enabling GPU Support

GPU support via GPU Manager and HAPI is not included by default when building Charm++. Use `build` with the `cuda` option to build Charm++ with GPU support (CMake build is currently not supported), e.g.

```
$ ./build charm++ netlrts-linux-x86_64 cuda -j8 --with-production
```

Building with GPU support requires an installation of the CUDA Toolkit on the system, which is automatically found by the build script. If the script fails to find it, provide the path as one of `CUDATOOLKIT_HOME`, `CUDA_DIR`, or `CUDA_HOME` environment variables.

Using GPU Support through HAPI

As explained in the Overview section, use of CUDA streams is strongly recommended. This provides the opportunity for kernels offloaded by chares to execute concurrently on the GPU.

In a typical Charm++ application using CUDA, the `.C` and `.ci` files would contain the Charm++ code, whereas a `.cu` file would contain the definition of CUDA kernels and functions that serve as entry points from the Charm++ application to use GPU capabilities. Calls to CUDA and/or HAPI to invoke kernels, perform data transfers, or enqueue

detection for GPU work completion would be placed inside this file, although they could also be put in the `.C` file provided that the right header files are included (`<cuda_runtime.h>` and/or `"hapi.h"`). The user should make sure that the CUDA kernels are compiled by `nvcc`, however.

After the necessary GPU operations are enqueued in the appropriate CUDA stream, the user would call `hapiAddCallback` to have a Charm++ callback to be invoked when all previous operations in the stream complete. For example, `hapiAddCallback` can be called after a kernel invocation and a device-to-host data transfer to asynchronously ‘enqueue’ a Charm++ callback that prints out the data computed by the GPU to `stdout`. The GPU Manager module ensures that the Charm++ callback will only be invoked once the GPU kernel and device-to-host data transfer complete.

The following is a list of HAPI functions:

```
void hapiAddCallback(cudaStream_t stream, CkCallback* callback);

void hapiCreateStreams();
cudaStream_t hapiGetStream();

void* hapiPoolMalloc(int size);
void hapiPoolFree(void* ptr);

hapiCheck(code);
```

`hapiCreateStreams` creates as many streams as the maximum number of concurrent kernels supported by the GPU device. `hapiGetStream` hands out a stream created by the runtime in a round-robin fashion. `hapiPool` functions provides memory pool functionalities which are used to obtain/free device memory without interrupting the GPU. `hapiCheck` is used to check if the input code block executes without errors. The provided code should return `cudaError_t` for it to work.

Examples using CUDA and HAPI can be found under `examples/charm++/cuda`. Codes under `#ifdef USE_WR` use the `hapiWorkRequest` scheme, which is now deprecated.

Direct GPU Messaging

Inspired by CUDA-aware MPI, the direct GPU messaging feature in Charm++ attempts to reduce the latency and increase the bandwidth for inter-GPU data transfers by bypassing host memory. In Charm++, however, some metadata exchanges between the sender and receiver are required as the destination buffer is not known by the sender. Thus our approach is based on the Zero Copy Entry Method Post API (section 2.3.1), where the sender sends a metadata message containing the address of the source buffer on the GPU and the receiver posts an `Rget` from the source buffer to the destination buffer (also on the GPU).

To send a GPU buffer using direct GPU messaging, add a `nocopydevice` specifier to the parameter of the receiving entry method in the `.ci` file:

```
entry void foo(int size, nocopydevice double arr[size]);
```

This entry method should be invoked on the sender by wrapping the source buffer with `CkDeviceBuffer`, whose constructor takes a pointer to the source buffer, a Charm++ callback to be invoked once the transfer completes (optional), and a CUDA stream associated with the transfer (also optional):

```
// Constructors of CkDeviceBuffer
CkDeviceBuffer(const void* ptr);
CkDeviceBuffer(const void* ptr, const CkCallback& cb);
CkDeviceBuffer(const void* ptr, cudaStream_t stream);
CkDeviceBuffer(const void* ptr, const CkCallback& cb, cudaStream_t stream);
```

(continues on next page)

(continued from previous page)

```
// Call on sender
someProxy.foo(source_buffer, cb, stream);
```

As with the Zero Copy Entry Method Post API, two entry methods (post entry method and regular entry method) must be specified, and the post entry method has an additional `CkDeviceBufferPost` argument that can be used to specify the CUDA stream where the data transfer will be enqueued:

```
// Post entry method
void foo(int& size, double*& arr, CkDeviceBufferPost* post) {
    arr = dest_buffer; // Inform the location of the destination buffer to the runtime
    post[0].cuda_stream = stream; // Perform the data transfer in the specified CUDA_
    ↪stream
}

// Regular entry method
void foo(int size, double* arr) {
    // Data transfer into arr has been initiated
    ...
}
```

The specified CUDA stream can be used by the receiver to asynchronously invoke GPU operations dependent on the arriving data, without explicitly synchronizing with the host. This brings us to an important difference from the host-side Zero Copy API: the regular entry method is invoked after the data transfer is **initiated**, not after it is complete. It should also be noted that the regular entry method can be defined as a SDAG method if so desired.

Currently the direct GPU messaging feature is limited to **intra-node** messages. Inter-node messages will be transferred using the naive host-staged mechanism where the data is first transferred to the host from the source GPU, sent over the network, then transferred to the destination GPU.

An optimized mechanism for inter-process communication using CUDA IPC, POSIX shared memory, and pre-allocated GPU communication buffers are available through runtime flags. This significantly reduces the overhead from creation and opening of CUDA IPC handles, especially for relatively small messages. `+gpushm` will turn on this optimization feature, `+gpcucommbuffer [size]` specifies the size of the communication buffer allocated on each GPU (default is 64MB), and `+gpuipc eventpool` determines the number of CUDA IPC events per PE that is used for this feature (default is 16).

Examples using the direct GPU messaging feature can be found in `examples/charm++/cuda/gpudirect`.

Intra-node Persistent GPU Communication

Persistent GPU communication is a feature designed to take advantage of the fact that the address and size of the source/destination buffers do not change in persistent communication. Such patterns are found in many applications, including iterative applications that exchange the same set of buffers between neighbors/peers at every iteration. This allows us to reduce the overheads involved with direct GPU messaging, including the nature of the multi-hop mechanism and CUDA IPC setup, and directly issue `cudaMemcpy` calls at the message send sites (for a `put` type operation; a `get` type operation is also supported). In inter-process communication, CUDA IPC handles are created, exchanged and opened only once (this setup step can be performed again if the buffer addresses or sizes involved in the persistent communication change). Note that this feature is currently supported only for intra-node communication.

Setup and communication routines are supported through `CkDevicePersistent`:

```
struct CkDevicePersistent {
    // Constructors
    CkDevicePersistent(const void* ptr, size_t size);
    CkDevicePersistent(const void* ptr, size_t size, const CkCallback& cb);
```

(continues on next page)

(continued from previous page)

```

CkDevicePersistent(const void* ptr, size_t size, cudaStream_t stream);
CkDevicePersistent(const void* ptr, size_t size, const CkCallback& cb, cudaStream_t
→stream);

void open(); // Creates an IPC handle for inter-process communication
void closes(); // Closes an opened IPC handle
void set_msg(void* msg); // Ties a Charm++ message to the stored callback
void get(CkDevicePersistent& src); // Initiates transfer from the specified source
→buffer
void put(CkDevicePersistent& dst); // Initiates transfer to the specified
→destination buffer
};

```

CkDevicePersistent objects should be created on both the sender and receiver chares, with the respective source and destination buffer addresses and sizes. The optional Charm++ callback object will be invoked once the communication using that persistent buffer is complete. For instance, if both sender and receiver CkDevicePersistent objects have callbacks associated with them, the sender's callback will be invoked once the send operation is complete and when the user can reuse/free the associated buffer, and the receiver's callback will be invoked once the received data is available. The associated CUDA stream will be used in the `cudaMemcpyAsync` calls to asynchronously invoke the underlying data transfers, which are tracked to invoke the subsequent callbacks, if any. The open and close calls are required for CUDA IPC setup and teardown. open should be called after the creation of the CkDevicePersistent object and before sending it to a communication peer, and close should be called when the persistent object will no longer be used or before migration.

Once the CkDevicePersistent objects are created and opened, they should be exchanged between the communication peers. After this setup stage, the objects' get and put methods can be invoked freely to perform the desired communication.

Examples using the intra-node persistent GPU communication can be found in `examples/charm++/cuda/gpudirect/persistent` and `examples/charm++/cuda/gpudirect/jacobi3d`.

2.3.15 Charm-MPI Interoperation

Codes and libraries written in Charm++ and MPI can also be used in an interoperable manner. Currently, this functionality is supported only if Charm++ is built using MPI, PAMILRTS, or GNI as the network layer (e.g. `mpi-linux-x86_64` build). An example program to demonstrate the interoperation is available in `examples/charm++/mpi-coexist`. In the following text, we will refer to this example program for the ease of understanding.

Control Flow and Memory Structure

The control flow and memory structure of a Charm++-MPI interoperable program is similar to that of a pure MPI program that uses external MPI libraries. The execution of program begins with pure MPI code's *main*. At some point after `MPI_Init()` has been invoked, the following function call should be made to initialize Charm++:

```
void CharmLibInit(MPI_Comm newComm, int argc, char **argv)
```

If Charm++ is build on top of MPI, `newComm` is the MPI communicator that Charm++ will use for the setup and communication. All the MPI ranks that belong to `newComm` should call this function collectively. A collection of MPI ranks that make the `CharmLibInit` call defines a new Charm++ instance. Different MPI ranks that belong to different communicators can make this call independently, and separate Charm++ instances that are not aware of each other will be created. This results in a space division. As of now, a particular MPI rank can only be part of one unique Charm++ instance. For PAMILRTS and GNI, the `newComm` argument is ignored. These layers do not support the space division of given processors, but require all processors to make the `CharmLibInit` call. The mode of interoperating here is called time division, and can be used with MPI-based Charm++ also if the size of `newComm` is

same as `MPI_COMM_WORLD`. Arguments `argc` and `argv` should contain the information required by Charm++ such as the load balancing strategy, etc.

During initialization, control is transferred from MPI to the Charm++ RTS on the MPI ranks that made the call. Along with basic setup, the Charm++ RTS also invokes the constructors of all mainchares during initialization. Once the initial set up is done, control is transferred back to MPI as if returning from a function call. Since Charm++ initialization is made via a function call from the pure MPI program, Charm++ resides in the same memory space as the pure MPI program. This makes transfer of data to/from MPI from/to Charm++ convenient (using pointers).

Writing Interoperable Charm++ Libraries

Minor modifications are required to make a Charm++ program interoperable with a pure MPI program:

- If the interoperable Charm++ library does not contain a main module, the Charm++ RTS provides a main module and the control is returned back to MPI after the initialization is complete. In the other case, the library should explicitly call `CkExit` to mark the end of initialization and the user should provide `-nomain-module` link time flag when building the final executable.
- `CkExit` should be used the same way a `return` statement is used for returning back from a function call. `CkExit` should be called only once from one of the processors. This unique call marks the transfer of control from Charm++ RTS to MPI.
- Include `mpi-interoperate.h` - if not included in the files that call `CkExit`, invoking `CkExit` will result in unspecified behavior.
- Since the `CharmLibInit` call invokes the constructors of mainchares, the constructors of mainchares should only perform basic set up such as creation of chare arrays etc, i.e. the set up should not result in invocation of actual work, which should be done using interface functions (when desired from the pure MPI program). However, if the main module is provided by the library, `CharmLibInit` behaves like a regular Charm++ startup and execution which is stopped when `CkExit` is explicitly called by the library. One can also avoid use of mainchares, and perform the necessary initializations in an interface function as demonstrated in the interoperable library `examples/charm++/mpi-coexist/libs/hello`.
- *Interface functions* - Every library needs to define interface function(s) that can be invoked from pure MPI programs, and transfers the control to the Charm++ RTS. The interface functions are simple functions whose task is to start work for the Charm++ libraries. Here is an example interface function for the *hello* library.

```
void HelloStart(int elems)
{
    if(CkMyPe() == 0) {
        CkPrintf("HelloStart - Starting lib by calling constructor of MainHello\n");
        CProxy_MainHello mainhello = CProxy_MainHello::ckNew(elems);
    }
    StartCharmScheduler(-1);
}
```

This function creates a new chare (`mainHello`) defined in the *hello* library which subsequently results in work being done in *hello* library. More examples of such interface functions can be found in `hi` (`HiStart`) and `kNeighbor` (`kNeighbor`) directories in `examples/charm++/mpi-coexist/libs`. Note that a scheduler call `StartCharmScheduler()` should be made from the interface functions to start the message reception by Charm++ RTS.

Writing Interoperable MPI Programs

An MPI program that invokes Charm++ libraries should include `mpi-interoperate.h`. As mentioned earlier, an initialization call, `CharmLibInit` is required after invoking `MPI_Init` to perform the initial set up of Charm++.

It is advisable to call an `MPI_Barrier` after a control transfer between Charm++ and MPI to avoid any side effects. Thereafter, a Charm++ library can be invoked at any point using the interface functions. One may look at `examples/charm++/mpi-coexist/multirun.cpp` for a working example. Based on the way interfaces are defined, a library can be invoked multiple times. In the end, one should call `CharmLibExit` to free resources reserved by Charm++.

Compilation and Execution

An interoperable Charm++ library can be compiled as usual using *charm*. Instead of producing an executable in the end, one should create a library (*.a) as shown in `examples/charm++/mpi-coexist/libs/hi/Makefile`. The compilation process of the MPI program, however, needs modification. One has to include the charm directory (`-I$(CHARMDIR)/include`) to help the compiler find the location of included `mpi-interoperate.h`. The linking step to create the executable should be done using *charm*, which in turn uses the compiler used to build charm. In the linking step, it is required to pass `-mpi` as an argument because of which *charm* performs the linking for interoperation. The charm libraries, which one wants to be linked, should be passed using `-module` option. Refer to `examples/charm++/mpi-coexist/Makefile` to view a working example. For execution on BG/Q systems, the following additional argument should be added to the launch command: `-envs PAMI_CLIENTS=MPI, Converse`.

User Driven Mode

In addition to the above technique for interoperation, one can also interoperate with Charm++ in user driven mode. User driven mode is intended for cases where the developer has direct control over the both the Charm++ code and the non-Charm++ code, and would like a more tightly coupled relation between the two. When executing in user driven mode, *main* is called on every rank as in the above example. To initialize the Charm++ runtime, a call to `CharmInit` should be called on every rank:

```
void CharmInit(int argc, char **argv)
```

`CharmInit` starts the Charm++ runtime in user driven mode, and executes the constructor of any main chares, and sends out messages for readonly variables and group creation. Control returns to user code after this initialization completes. Once control is returned, user code can do other work as needed, including creating chares, and invoking entry methods on proxies. Any messages created by the user code will be sent/received the next time the user calls `StartCharmScheduler`. Calls to `StartCharmScheduler` allow the Charm++ runtime to resume sending and processing messages, and control returns to user code when `CkExit` is called. The Charm++ scheduler can be started and stopped in this fashion as many times as necessary. `CharmLibExit` should be called by the user code at the end of execution to exit the entire application.

Applications which wish to use readonlies, and/or create groups before the rest of the application runs without using a mainchare can do a split initialization. `CharmBeginInit` initializes the runtime system and immediately returns control to the user after any mainchares are created. At this point, user code can create groups, and set readonly variables on PE 0. Then, a call to `CharmFinishInit` does the rest of the initialization by sending out messages from PE 0 to the rest of the PEs for creating groups and setting readonlies. `CharmInit` is just a shortcut for calling these two functions one after another.

```
void CharmBeginInit(int argc, char **argv) void CharmFinishInit()
```

A small example of user driven interoperation can be found in `examples/charm++/user-driven-interop`.

2.3.16 Interoperation with Kokkos

Kokkos is a shared-memory parallel programming model in C++ developed by Sandia National Laboratories (<https://github.com/kokkos/kokkos>). It aims to provide ‘performance portability’ to HPC applications through abstractions for parallel execution and data management. For execution in distributed memory environments, however, other

frameworks such as MPI must be used in conjunction, to enable multiple Kokkos processes running on potentially different physical nodes to communicate with each other.

In this section, we explore the basic interoperability of Kokkos with Charm++. Currently there is no sophisticated integration scheme, Charm++ only manages the communication between different Kokkos instances with each instance individually managing the parallel execution underneath. Example programs can be found in `examples/charm++/shared_runtimes/kokkos/hello` and `examples/charm++/shared_runtimes/kokkos/vecadd`.

Compiling the Kokkos Library

Kokkos supports multiple backends for parallel execution. We recommend OpenMP for multicore CPUs and CUDA for machines with GPUs. Because Kokkos can be built with more than one backend, it is preferable to build both OpenMP and CUDA backends on GPU machines.

To build Kokkos with the OpenMP backend, run the following commands from the Kokkos source folder:

```
$ mkdir build-omp
$ cd build-omp
$ ../generate_makefile.bash --prefix=<absolute path to build-omp> --with-openmp
                        --arch=<CPU architecture>
$ make -j kokkoslib
$ make install
```

To build Kokkos with both OpenMP and CUDA backends (required for `vecadd` example), use the following commands:

```
$ mkdir build-cuda
$ cd build-cuda
$ generate_makefile.bash --prefix=<absolute path to build-cuda>
                        --with-cuda=<path to CUDA toolkit>
                        --with-cuda-options=enable_lambda
                        --with-openmp --arch=<CPU arch>,<GPU arch>
                        --compiler=<path to included NVCC wrapper>
$ make -j kokkoslib
$ make install
```

For more compilation options, please refer to <https://github.com/kokkos/kokkos/wiki/Compiling>.

Program Structure and Flow

The basic programming pattern using Kokkos and Charm++ together for parallel execution in distributed memory environments is the following. We use a Charm++ nodegroup (which corresponds to a OS process) to encapsulate a Kokkos instance that will manage the parallel execution underneath. We initialize Kokkos using `Kokkos::initialize()` in the constructor of the nodegroup, and finalize it using `Kokkos::finalize()`. Calls to the Kokkos parallel API such as `Kokkos::parallel_for()` can be made between these calls. Communication between the different Kokkos instances can be done via messages and entry method invocation among the nodegroup chares as in regular Charm++.

2.3.17 Interoperation with RAJA

RAJA is a shared-memory parallel programming model in C++ developed by Lawrence Livermore National Laboratory (<https://github.com/LLNL/RAJA>). RAJA shares similar goals and concepts with Kokkos (Section 2.3.16).

In this section, we explore the basic interoperability of RAJA with Charm++. Currently there is no sophisticated integration scheme, Charm++ only manages the communication between different RAJA instances with each instance

individually managing the parallel execution underneath. Example programs can be found in `examples/charm++/shared_runtimes/raja/hello` and `examples/charm++/shared_runtimes/raja/vecadd`.

Compiling the RAJA Library

RAJA supports multiple backends for parallel execution. We recommend OpenMP for multicore CPUs and CUDA for machines with GPUs. Because RAJA can be built with more than one backend, it is preferable to build both OpenMP and CUDA backends on GPU machines.

To build RAJA with both OpenMP and CUDA backends (required for `vecadd` example), use the following commands:

```
$ mkdir build && install
$ cd build
$ cmake -DENABLE_CUDA=On -DCMAKE_INSTALL_PREFIX=<path to RAJA install folder> ../
$ make -j
$ make install
```

For more compilation options and assistance, please refer to the [RAJA User Guide](#).

Program Structure and Flow

The basic programming pattern using RAJA and Charm++ together for parallel execution in distributed memory environments is the following. We use a Charm++ nodegroup (which corresponds to a OS process) to encapsulate a RAJA instance that will manage the parallel execution underneath. Calls to the RAJA parallel API such as `RAJA::forall()` can be made in a method of the nodegroup to perform parallel computation in shared memory. Communication between the different RAJA instances can be performed via messages and entry method invocation among the nodegroup chares as in regular Charm++.

2.3.18 Partitioning in Charm++

Starting with the 6.5.0 release, Charm++ was augmented with support for partitioning. The key idea is to divide the allocated set of nodes into subsets that run independent Charm++ instances. These Charm++ instances (called *partitions* from now on) have a unique identifier, can be programmed to do different tasks, and can interact with each other. Addition of the partitioning scheme does not affect the existing code base or codes that do not want to use partitioning. Some of the use cases of partitioning are replicated NAMD, replica-based fault tolerance, studying mapping performance etc. In some aspects, partitioning is similar to disjoint communicator creation in MPI.

Overview

The Charm++ stack has three components - Charm++, Converse and a machine layer. In general, machine layer handles the exchange of messages among nodes, and interacts with the next layer in the stack - Converse. Converse is responsible for scheduling of tasks (including user code) and is used by Charm++ to execute the user application. Charm++ is the top-most level in which the applications are written. During partitioning, Charm++ and machine layers are unaware of the partitioning. Charm++ assumes its partition to be the entire world, whereas machine layer considers the whole set of allocated nodes as one partition. During start up, converse divides the allocated set of nodes into partitions, in each of which Charm++ instances are run. It performs the necessary translations as interactions happen with Charm++ and the machine layer. The partitions can communicate with each other using the Converse API described later.

Ranking

The Charm++ stack assigns a rank to every processing element (PE). In the non-partitioned version, a rank assigned to a PE is the same at all three layers of the Charm++ stack. This rank also generally coincides with the rank provided to processors/cores by the underlying job scheduler. The importance of these ranks derive from the fact that they are used for multiple purposes. Partitioning leads to segregation of the notion of ranks at different levels of Charm++ stack. What used to be the PE is now a local rank within a partition running a Charm++ instance. Existing methods such as `CkMyPe()`, `CkMyNode()`, `CmiMyPe()`, etc. continue to provide these local ranks. Hence, existing codes do not require any change as long as inter-partition interaction is not required.

On the other hand, machine layer is provided with the target ranks that are globally unique. These ranks can be obtained using functions with the *Global* suffix such as `CmiNumNodesGlobal()`, `CmiMyNodeGlobal()`, `CmiMyPeGlobal()` etc.

Converse, which operates at a layer between Charm++ and machine layer, performs the required transitions. It maintains relevant information for any conversion. Information related to partitions can be obtained using Converse level functions such as `CmiMyPartition()`, `CmiNumPartitions()`, etc. If required, one can also obtain the mapping of a local rank to a global rank using functions such as `CmiGetPeGlobal(int perank, int partition)` and `CmiGetNodeGlobal(int noderank, int partition)`. These functions take two arguments - the local rank and the partition number. For example, `CmiGetNodeGlobal(5, 2)` will return the global rank of the node that belongs to partition 2 and has a local rank of 5 in partition 2. The inverse translation, from global rank to local rank, is not supported.

Startup and Partitioning

A number of compile time and runtime parameters are available for users who want to run multiple partitions in one single job.

- Runtime parameter: `+partitions <part_number>` or `+replicas <replica_number>` - number of partitions to be created. If no further options are provided, allocated cores/nodes are divided equally among partitions. Only this option is supported from the 6.5.0 release; remaining options are supported starting 6.6.0.
- Runtime parameter: `+master_partition` - assign one core/node as the master partition (partition 0), and divide the remaining cores/nodes equally among remaining partitions.
- Runtime parameter: `+partition_sizes L[-U[:S[.R]]]#W[, ...]` - defines the size of partitions. A single number identifies a particular partition. Two numbers separated by a dash identify an inclusive range (*lower bound* and *upper bound*). If they are followed by a colon and another number (a *stride*), that range will be stepped through in increments of the additional number. Within each stride, a dot followed by a *run* will indicate how many partitions to use from that starting point. Finally, a compulsory number sign (#) followed by a *width* defines the size of each of the partitions identified so far. For example, the sequence `0-4:2#10, 1#5, 3#15` states that partitions 0, 2, 4 should be of size 10, partition 1 of size 5 and partition 3 of size 15. In SMP mode, these sizes are in terms of nodes. All workers threads associated with a node are assigned to the partition of the node. This option conflicts with `+assign_master`.
- Runtime parameter: `+partition_topology` - use a default topology aware scheme to partition the allocated nodes.
- Runtime parameter: `+partition_topology_scheme <scheme>` - use the given scheme to partition the allocated nodes. Currently, two generalized schemes are supported that should be useful on torus networks. If scheme is set to 1, allocated nodes are traversed plane by plane during partitioning. A hilbert curve based traversal is used with scheme 2.
- Compilation parameter: `-custom-part`, runtime parameter: `+use_custom_partition` - enables use of user defined partitioning. In order to implement a new partitioning scheme, a user must link an object exporting a C function with following prototype:

```
extern "C" void createCustomPartitions(int numparts, int *partitionSize,
int *nodeMap);
```

`numparts` (input) - number of partitions to be created.

`partitionSize` (input) - an array that contains the size of each partition.

`nodeMap` (output, preallocated) - a preallocated array of length `CmiNumNodesGlobal()`. Entry i in this array specifies the new global node rank of a node with default node rank i . The entries in this array are block-wise divided to create partitions, i.e. entries 0 to `partitionSize[0]-1` belong to partition 1, `partitionSize[0]` to `partitionSize[0]+partitionSize[1]-1` to partition 2 and so on.

When this function is invoked to create partitions, `TopoManager` is configured to view all the allocated nodes as one partition. Partition based API is yet to be initialized, and should not be used. A link time parameter `-custom-part` is required to be passed to `charmcc` for successful compilation.

Redirecting output from individual partitions

Output to standard output (stdout) from various partitions can be directed to separate files by passing the target path as a command line option. The run time parameter `+stdout <path>` is to be used for this purpose. The `<path>` may contain the C format specifier `%d`, which will be replaced by the partition number. In case, `%d` is specified multiple times, only the first three instances from the left will be replaced by the partition number (other or additional format specifiers will result in undefined behavior). If a format specifier is not specified, the partition number will be appended as a suffix to the specified path. Example usage:

- `+stdout out/%d/log` will write to `out/0/log`, `out/1/log`, `out/2/log`, ...
- `+stdout log` will write to `log.0`, `log.1`, `log.2`, ...
- `+stdout out/%d/log%d` will write to `out/0/log0`, `out/1/log1`, `out/2/log2`, ...

Inter-partition Communication

A new API was added to Converse to enable sending messages from one replica to another. Currently, the following functions are available:

- `CmiInterSyncSend(local_rank, partition, size, message)`
- `CmiInterSyncSendAndFree(local_rank, partition, size, message)`
- `CmiInterSyncNodeSend(local_node, partition, size, message)`
- `CmiInterSyncNodeSendAndFree(local_node, partition, size, message)`

Users who have coded in Converse will find these functions to be very similar to basic Converse functions for send - `CmiSyncSend` and `CmiSyncSendAndFree`. Given the local rank of a PE and the partition it belongs to, these two functions will pass the message to the machine layer. `CmiInterSyncSend` does not return until message is ready for reuse. `CmiInterSyncSendAndFree` passes the ownership of message to the Charm++ RTS, which will free the message when the send is complete. Each converse message contains a message header, which makes those messages active - they contain information about their handlers. These handlers can be registered using existing API in Charm++ - `CmiRegisterHandler`. `CmiInterNodeSend` and `CmiInterNodeSendAndFree` are counterparts to these functions that allow sending of a message to a node (in SMP mode).

2.4 Expert-Level Functionality

2.4.1 Tuning and Developing Load Balancers

Load Balancing Simulation

The simulation feature of the load balancing framework allows the users to collect information about the compute WALL/CPU time and communication of the chares during a particular run of the program and use this information later to test the different load balancing strategies to see which one is suitable for the program behavior. Currently, this feature is supported only for the centralized load balancing strategies. For this, the load balancing framework accepts the following command line options:

1. *+LBDump StepStart*

This will dump the compute and the communication data collected by the load balancing framework starting from the load balancing step *StepStart* into a file on the disk. The name of the file is given by the *+LBDumpFile* option. The load balancing step in the program is numbered starting from 0. Negative value for *StepStart* will be converted to 0.

2. *+LBDumpSteps StepsNo*

This option specifies the number of load balancing steps for which data will be dumped to disk. If omitted, its default value is 1. The program will exit after *StepsNo* files are created.

3. *+LBDumpFile FileName*

This option specifies the base name of the file created with the load balancing data. If this option is not specified, the framework uses the default file `lbdata.dat`. Since multiple steps are allowed, a number corresponding to the step number is appended to the filename in the form `Filename.#`; this applies to both dump and simulation.

4. *+LBSim StepStart*

This option instructs the framework to do the simulation starting from *StepStart* step. When this option is specified, the load balancing data along with the step number will be read from the file specified in the *+LBDumpFile* option. The program will print the results of the balancing for a number of steps given by the *+LBSimSteps* option, and then will exit.

5. *+LBSimSteps StepsNo*

This option is applicable only to the simulation mode. It specifies the number of load balancing steps for which the data will be dumped. The default value is 1.

6. *+LBSimProcs*

With this option, the user can change the number of processors specified to the load balancing strategy. It may be used to test the strategy in the cases where some processor crashes or a new processor becomes available. If this number is not changed since the original run, starting from the second step file, the program will print other additional information about how the simulated load differs from the real load during the run (considering all strategies that were applied while running). This may be used to test the validity of a load balancer prediction over the reality. If the strategies used during run and simulation differ, the additional data printed may not be useful.

Here is an example which collects the data for a 1000 processor run of a program

```
$ ./charmrun pgm +p1000 +balancer RandCentLB +LBDump 2 +LBDumpSteps 4 +LBDumpFile_
↳ lbsim.dat
```

This will collect data on files `lbsim.dat.2,3,4,5`. We can use this data to analyze the performance of various centralized strategies using:

```
$ ./charmrun pgm +balancer <Strategy to test> +LBSim 2 +LBSimSteps 4 +LBDumpFile_
↳ lbsim.dat
[+LBSimProcs 900]
```

Please note that this does not invoke the real application. In fact, “pgm” can be replaced with any generic application which calls centralized load balancer. An example can be found in `tests/charm++/load_balancing/`

lb_test.

Future load predictor

When objects do not follow the assumption that the future workload will be the same as the past, the load balancer might not have the right information to do a good rebalancing job. To prevent this, the user can provide a transition function to the load balancer to predict what will be the future workload, given the past instrumented one. For this, the user can provide a specific class which inherits from `LBPredictorFunction` and implement the appropriate functions. Here is the abstract class:

```
class LBPredictorFunction {
public:
    int num_params;

    virtual void initialize_params(double *x);

    virtual double predict(double x, double *params) =0;
    virtual void print(double *params) {PredictorPrintf("LB: unknown model");};
    virtual void function(double x, double *param, double &y, double *dyda) =0;
};
```

- `initialize_params` by default initializes the parameters randomly. If the user knows how they should be, this function can be re-implemented.
- `predict` is the function that predicts the future load based on the function parameters. An example for the `predict` function is given below.

```
double predict(double x, double *param) {return (param[0]*x + param[1]);}
```

- `print` is useful for debugging and it can be re-implemented to have a meaningful print of the learned model
- `function` is a function internally needed to learn the parameters, `x` and `param` are input, `y` and `dyda` are output (the computed function and all its derivatives with respect to the parameters, respectively). For the function in the example should look like:

```
void function(double x, double *param, double &y, double *dyda) {
    y = predict(x, param);
    dyda[0] = x;
    dyda[1] = 1;
}
```

Other than these functions, the user should provide a constructor which must initialize `num_params` to the number of parameters the model has to learn. This number is the dimension of `param` and `dyda` in the previous functions. For the given example, the constructor is `{num_params = 2;}`.

If the model for computation is not known, the user can leave the system to use the default function.

As seen, the function can have several parameters which will be learned during the execution of the program. For this, user can be add the following command line arguments to specify the learning behavior:

1. *+LBPredictorWindow size*

This parameter specifies the number of statistics steps the load balancer will store. The greater this number is, the better the approximation of the workload will be, but more memory is required to store the intermediate information. The default is 20.

2. *+LBPredictorDelay steps*

This will tell how many load balancer steps to wait before considering the function parameters learned and starting to use the mode. The load balancer will collect statistics for a `+LBPredictorWindow` steps, but it will start using the model as soon as `+LBPredictorDelay` information are collected. The default is 10.

Moreover, another flag can be set to enable the predictor from command line: `+LBPredictor`.

Other than the command line options, there are some methods which can be called from the user program to modify the predictor. These methods are:

- `void PredictorOn(LBPredictorFunction *model);`
- `void PredictorOn(LBPredictorFunction *model, int window);`
- `void PredictorOff();`
- `void ChangePredictor(LBPredictorFunction *model);`

An example can be found in `tests/charm++/load_balancing/lb_test/predictor`.

Control CPU Load Statistics

Charm++ programmers can modify the CPU load data in the load balancing database before a load balancing phase starts (which is the time when load balancing database is collected and used by load balancing strategies).

In an array element, the following function can be invoked to overwrite the CPU load that is measured by the load balancing framework.

```
double newTiming;  
setObjTime(newTiming);
```

`setObjTime()` is defined as a method of class `CkMigratable`, which is the superclass of all array elements.

The users can also retrieve the current timing that the load balancing runtime has measured for the current array element using `getObjTime()`.

```
double measuredTiming;  
measuredTiming = getObjTime();
```

This is useful when the users want to derive a new CPU load based on the existing one.

Model-based Load Balancing

The user can choose to feed load balancer with their own CPU timing for each Chare based on certain computational model of the applications.

To do so, in the array element's constructor, the user first needs to turn off automatic CPU load measurement completely by setting

```
usesAutoMeasure = false;
```

The user must also implement the following function to the chare array classes:

```
virtual void CkMigratable::UserSetLBLoad(); // defined in base class
```

This function serves as a callback that is called on each chare object when `AtSync()` is called and ready to do load balancing. The implementation of `UserSetLBLoad()` is simply to set the current chare object's CPU load in load balancing framework. `setObjTime()` described above can be used for this.

Writing a new load balancing strategy

Charm++ programmers can choose an existing load balancing strategy from Charm++'s built-in strategies (see 2.2.6) for the best performance based on the characteristics of their applications. However, they can also choose to write their own load balancing strategies.

Users are recommended to use the TreeLB structure to write new custom strategies if possible (see 2.4.1). Currently, TreeLB may not be suitable for distributed strategies, if communication graph information is needed, or when working with legacy strategies.

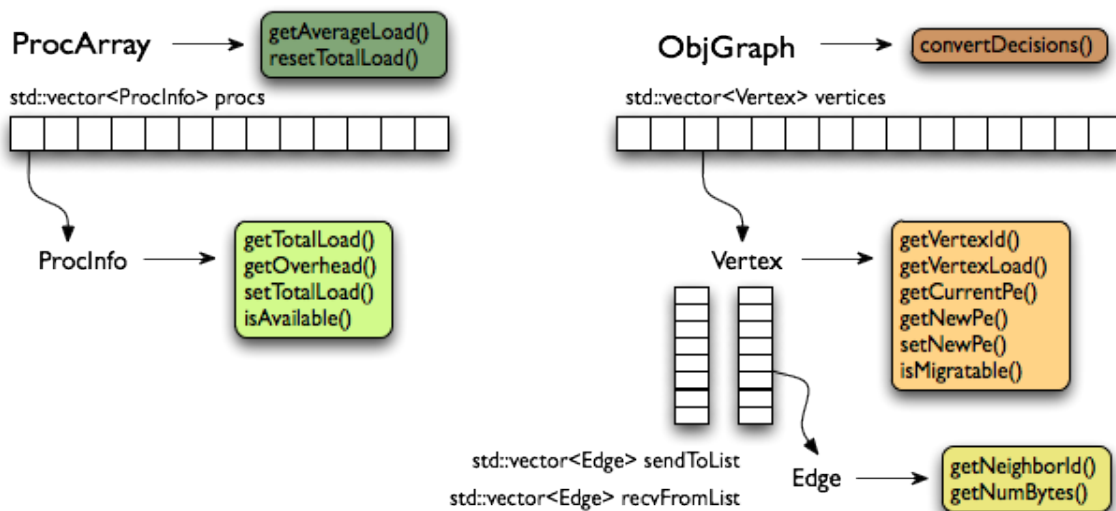
The Charm++ load balancing framework provides a simple scheme to incorporate new load balancing strategies. The programmer needs to write their strategy for load balancing based on the instrumented ProcArray and ObjGraph provided by the load balancing framework. This strategy is implemented within this function:

```
void FooLB::work(LDStats *stats) {
    /** ===== INITIALIZATION ===== */
    ProcArray *parr = new ProcArray(stats);
    ObjGraph *ogr = new ObjGraph(stats);

    /** ===== STRATEGY ===== */
    /// The strategy goes here
    /// The strategy goes here
    /// The strategy goes here
    /// The strategy goes here
    /// The strategy goes here

    /** ===== CLEANUP ===== */
    ogr->convertDecisions(stats);
}
```

Figure 2 explains the two data structures available to the strategy: ProcArray and ObjGraph. Using them, the strategy should assign objects to new processors where it wants to be migrated through the setNewPe() method. src/ck-ldb/GreedyLB.C can be referred.



2: ProcArray and ObjGraph data structures to be used when writing a load balancing strategy

Use your own load balancer

Compile it in the form of a library and name it *libmoduleFooLB.a* where *FooLB* is the new load balancer. Add the path to the library and link the load balancer into an application using *-module FooLB*.

You can create a library in the following way. This will create *libmoduleFooLB.a*.

```
$ bin/charmc -o libmoduleFooLB.a FooLB.C
```

To include this balancer in your application, the application's Makefile can be changed in the following way:

```
$(TARGET): $(OBJECTS)
    $(CHARMC) -o $(TARGET) -L/path-to-the-lib $(OBJS) -module FooLB
```

Incorporating this strategy into the Charm++ build framework is explained in the next section.

Adding a load balancer to Charm++

Let us assume that we are writing a new centralized load balancer called *FooLB*. The next few steps explain the steps of adding the load balancer to the Charm++ build system:

1. Create files named *FooLB.ci*, *FooLB.h* and *FooLB.C* in directory of *src/ck-ldb*. One can choose to copy and rename the files *GraphPartLB.** and rename the class name in those files.
2. Implement the strategy in the *FooLB* class method — **FooLB::work(LDStats* stats)** as described in the previous section.
3. Build charm for your platform (This will create the required links in the tmp directory).
4. To compile the strategy files, first add *FooLB* in the *ALL_LDBS* list in *charm/tmp/Makefile_lb.sh*. Also comment out the line containing *UNCOMMON_LDBS* in *Makefile_lb.sh*. If *FooLB* will require some libraries at link time, you also need to create the dependency file called *libmoduleFooLB.dep*. Run the script in *charm/tmp*, which creates the new Makefile named "Make.lb".
5. Run `make depends` to update dependence rule of Charm++ files. And run `make charm++` to compile Charm++ which includes the new load balancing strategy files.

Writing a new load balancing strategy with TreeLB

Writing a load balancing strategy with TreeLB is very simple. It involves implementing a class inheriting from the abstract *Strategy* class defined in the *TreeStrategyBase.h* header file. Specifically, the load balancing strategy needs to be implemented in the *solve* method, which receives objects and processors as vectors. The solution needs to be stored in the *solution* object using its *assign(O object, P pe)* instance method and is used by the load balancing framework to perform object migrations accordingly. The *bool objSorted* argument specifies whether the incoming vector of objects is sorted by load values or not. This new strategy can be written in a header file and must be included in the *TreeStrategyFactory.h* header file along with the mapping of the strategy name and class to be invoked for the newly implemented strategy. It can then be used by linking a program with TreeLB and specifying the newly created strategy in the configuration file.

```
namespace TreeStrategy
{
    template <typename O, typename P, typename S>
    class FooLB : public Strategy<O, P, S>
    {
    public:
        void solve(std::vector<O>& objs, std::vector<P>& procs, S& solution, bool_
        ↪objSorted)
```

(continues on next page)

(continued from previous page)

```

{
    // The strategy goes here.
    // This is a simple example strategy that round-robin assigns objects to
    ↪processors.
    int index = 0;
    for (const auto& o : objs)
    {
        P p = procs[index];
        solution.assign(o, p); // update solution object
        index = (index + 1) % procs.size();
    }
};
} // namespace TreeStrategy

```

Understand Load Balancing Database Data Structure

To write a load balancing strategy, you need to know what information is measured during the runtime and how it is represented in the load balancing database data structure.

There are mainly 3 categories of information: a) processor information including processor speed, background load; b) object information including per object CPU/WallClock compute time and c) communication information .

The database data structure named LDStats is defined in *CentralLB.h*:

```

struct ProcStats { // per processor
    LBRealType total_walltime;
    LBRealType total_cputime;
    LBRealType idletime;
    LBRealType bg_walltime;
    LBRealType bg_cputime;
    int pe_speed;
    double utilization;
    bool available;
    int n_objs;
}

struct LDStats { // load balancing database
    ProcStats *procs;
    int count;

    int n_objs;
    int n_migrateobjs;
    LDObjData* objData;

    int n_comm;
    LDCommData* commData;

    int *from_proc, *to_proc;
}

```

1. *LBRealType* is the data type for load balancer measured time. It is “double” by default. User can specify the type to float at Charm++ compile time if want. For example, `./build charm++ netlrts-linux-x86_64 -with-lbtime-type=float`;
2. *procs* array defines processor attributes and usage data for each processor;

3. *objData* array records per object information, *LDObjData* is defined in *lbdb.h*;
4. *commData* array records per communication information. *LDCommData* is defined in *lbdb.h*.

2.4.2 Dynamic Code Injection

The Python scripting language in Charm++ allows the user to dynamically execute pieces of code inside a running application, without the need to recompile. This is performed through the CCS (Converse Client Server) framework (see *Converse Client-Server Interface* for more information about this). The user specifies which elements of the system will be accessible through the interface, as we will see later, and then run a client which connects to the server.

In order to exploit this functionality, Python interpreter needs to be installed into the system, and Charm++ LIBS need to be built with: `./build LIBS <arch> <options>`

The interface provides three different types of requests:

Execute requests to execute a code, it will contain the code to be executed on the server, together with the instructions on how to handle the environment;

Print asks the server to send back all the strings which have been printed by the script until now;

Finished asks the server if the current script has finished or it is still running.

There are three modes to run code on the server, ordered here by increase of functionality, and decrease of dynamic flexibility:

- **simple read/write** By implementing the read and write methods of the object exposed to python, in this way single variables may be exposed, and the code will have the possibility to modify them individually as desired. (see section 2.4.2)
- **iteration** By implementing the iterator functions in the server (see 2.4.2), the user can upload the code of a Python function and a user-defined iterator structure, and the system will apply the specified function to all the objects reflected by the iterator structure.
- **high level** By implementing python entry methods, the Python code uploaded can access them and activate complex, parallel operations that will be performed by the Charm++ application. (see section 2.4.2)

This documentation will describe the client API first, and then the server API.

Client API

In order to facilitate the interface between the client and the server, some classes are available to the user to include into the client. Currently C++ and java interfaces are provided.

C++ programs need to include `PythonCCS-client.h` into their code. This file is among the Charm++ include files. For java, the package `charm.ccs` needs to be imported. This is located under the java directory on the Charm++ distribution, and it provides both the Python and CCS interface classes.

There are three main classes provided: `PythonExecute`, `PythonPrint`, and `PythonFinished` which are used for the three different types of request.

All of them have two common methods to enable communication across different platforms:

int size(); Returns the size of the class, as number of bytes that will be transmitted through the network (this includes the code and other dynamic variables in the case of `PythonExecute`).

char *pack(); Returns a new memory location containing the data to be sent to the server, this is the data which has to be passed to the `CcsSendRequest` function. The original class will be unmodified and can be reused in subsequent calls.

A typical invocation to send a request from the client to the server has the following format:

```
CcsSendRequest (&server, "pyCode", 0, request.size(), request.pack());
```

PythonExecute

To execute a Python script on a running server, the client has to create an instance of `PythonExecute`, the two constructors have the following signature (java has a corresponding functionality):

```
PythonExecute(char *code, bool persistent=false, bool highlevel=false, CmiUInt4 _
↪interpreter=0);
PythonExecute(char *code, char *method, PythonIterator *info, bool persistent=false,
              bool highlevel=false, CmiUInt4 interpreter=0);
```

The second one is used for iterative requests (see 2.4.2). The only required argument is the code, a null terminated string, which will not be modified by the system. All the other parameters are optional. They refer to the possible variants for an execution request. In particular, this is a list of all the options:

iterative If the request is a single code (false) or if it represents a function over which to iterate (true) (see 2.4.2 for more details).

persistent It is possible to store information on the server which will be retained across different client calls (from simple data all the way up to complete libraries). True means that the information will be retained on the server, false means that the information will be deleted when the script terminates. In order to properly release the memory, when the last call is made (and the data is no longer required), this flag should be set to false. To reuse persistent data, the interpreter field of the request should be set to handle returned by a previous persistent call (see later in this subsection).

high level In order to have the ability to call high level Charm++ functions (available through the keyword python) this flag must be set to true. If it is false, the entire module “charm” will not be present, but the startup of the script will be faster.

print retain When the requested action triggers printed output for the client, this data can be retrieved with a Python-Print request. If the output is not desired, this flag can be set to false, and the output will be discarded. If it is set to true the output will be buffered pending retrieval by the client. The data will survive also after the termination of the Python script, and if not retrieved will bloat memory usage on the server.

busy waiting Instead of returning a handle immediately to the client, that can be used to retrieve prints and check if the script has finished, the server will answer to the client only when the script has terminated to run (and it will effectively work as a PythonFinished request).

These flags can be set and checked with the following routines (CmiUInt4 represent a 4 byte unsigned integer):

```
void setCode(char *set);
void setPersistent(bool set);
void setIterate(bool set);
void setHighLevel(bool set);
void setKeepPrint(bool set);
void setWait(bool set);
void setInterpreter(CmiUInt4 i);

bool isPersistent();
bool isIterate();
bool isHighLevel();
bool isKeepPrint();
bool isWait();
CmiUInt4 getInterpreter();
```

From a PythonExecute request, the server will answer with a 4 byte integer value, which is a handle for the interpreter that is running. It can be used to request for prints, check if the script has finished, and for reusing the same interpreter (if it was persistent).

A value of 0 means that there was an error and the script didn't run. This is typically due to a request to reuse an existing interpreter which is not available, either because it was not persistent or because another script is still running on that interpreter.

Auto-imported modules

When a Python script is run inside a Charm++ application, two Python modules are made available by the system. One is **ck**, the other is **charm**. The first one is always present and it represent basic functions, the second is related to high level scripting and it is present only when this is enabled (see 2.4.2 for how to enable it, and 2.4.2 for a description on how to implement charm functions).

The methods present in the **ck** module are the following:

printstr It accepts a string as parameter. It will write into the server stdout that string using the `CkPrintf` function call.

printclient It accepts a string as parameter. It will forward the string back to the client when it issues a `PythonPrint` request. It will buffer the strings until requested by `PythonPrint` if the `KeepPrint` option is true, otherwise it will discard them.

mype Requires no parameters, and will return an integer representing the current processor where the code is executing. It is equivalent to the Charm++ function `CkMyPe()`.

numpes Requires no parameters, and will return an integer representing the total number of processors that the application is using. It is equivalent to the Charm++ function `CkNumPes()`.

myindex Requires no parameters, and will return the index of the current element inside the array, if the object under which Python is running is an array, or None if it is running under a Chare, a Group or a Nodegroup. The index will be a tuple containing as many numbers as the dimension of the array.

read It accepts one object parameter, and it will perform a read request to the Charm++ object connected to the Python script, and return an object containing the data read (see 2.4.2 for a description of this functionality). An example of a call can be: `value = ck.read((number, param, var2, var3))` where the double parenthesis are needed to create a single tuple object containing four values passed as a single parameter, instead of four different parameters.

write It accepts two object parameters, and it will perform a write request to the Charm++ object connected to the Python script. For a description of this method, see 2.4.2. Again, only two objects need to be passed, so extra parenthesis may be needed to create tuples from individual values.

Iterate mode

Sometimes some operations need to be iterated over all the elements in the system. This “iterative” functionality provides a shortcut for the client user to do this. As an example, suppose we have a system which contains particles, with their position, velocity and mass. If we implement `read` and `write` routines which allow us to access single particle attributes, we may upload a script which doubles the mass of the particles with velocity greater than 1:

```
size = ck.read(("numparticles", 0));
for i in range(0, size):
    vel = ck.read(("velocity", i));
    mass = ck.read(("mass", i));
    mass = mass * 2;
    if (vel > 1): ck.write(("mass", i), mass);
```

Instead of all these read and writes, it will be better to be able to write:

```
def increase(p):
    if (p.velocity > 1): p.mass = p.mass * 2;
```

This is what the “iterative” functionality provides. In order for this to work, the server has to implement two additional functions (see 2.4.2), and the client has to pass some more information together with the code. This information is the name of the function that has to be called (which can be defined in the “code” or was previously uploaded to a persistent interpreter), and a user defined structure which specifies over what data the function should be invoked. These values can be specified either while constructing the `PythonExecute` variable (see the second constructor in section 2.4.2), or with the following methods:

```
void setMethodName(char *name);
void setIterator(PythonIterator *iter);
```

The `PythonIterator` object must be defined by the user, and the user must insure that the same definition is present inside both the client and the server. The Charm++ system will simply pass this structure as a void pointer. This structure must inherit from `PythonIterator`. In the simple case (highly recommended), wherein no pointers or dynamic allocation are used inside this class, nothing else needs to be done because it is trivial to serialize such objects.

If instead pointers or dynamic memory allocation are used, the following methods have to be reimplemented to support correct serialization:

```
int size();
char * pack();
void unpack();
```

The first returns the size of the class/structure after being packed. The second returns a pointer to a newly allocated memory containing all the packed data, the returned memory must be compatible with the class itself, since later on this same memory a call to `unpack` will be performed. Finally, the third will do the work opposite to `pack` and fix all the pointers. This method will not return anything and is supposed to fix the pointers “inline”.

PythonPrint

In order to receive the output printed by the Python script, the client needs to send a `PythonPrint` request to the server. The constructor is:

```
PythonPrint(CmiUInt4 interpreter, bool Wait=true, bool Kill=false);
```

The interpreter for which the request is made is mandatory. The other parameters are optional. The wait parameter represents whether a reply will be sent back immediately to the client even if there is no output (false), or if the answer will be delayed until there is an output (true). The kill option set to true means that this is not a normal request, but a signal to unblock the latest print request which was blocking.

The returned data will be a non null-terminated string if some data is present (or if the request is blocking), or a 4 byte zero data if nothing is present. This zero reply can happen in different situations:

- If the request is non blocking and no data is available on the server;
- If a kill request is sent, the previous blocking request is squashed;
- If the Python code ends without any output and it is not persistent;
- If another print request arrives, the previous one is squashed and the second one is kept.

As for a print kill request, no data is expected to come back, so it is safe to call `CcsNoResponse(server)`.

The two options can also be dynamically set with the following methods:

```
void setWait(bool set);
bool isWait();

void setKill(bool set);
bool isKill();
```

PythonFinished

In order to know when a Python code has finished executing, especially when using persistent interpreters, and a serialization of the scripts is needed, a `PythonFinished` request is available. The constructor is the following:

PythonFinished(CmiUInt4 interpreter, bool Wait=true);

The interpreter corresponds to the handle for which the request was sent, while the wait option refers to a blocking call (true), or immediate return (false).

The wait option can be dynamically modified with the two methods:

```
void setWait(bool set);
bool isWait();
```

This request will return a 4 byte integer containing the same interpreter value if the Python script has already finished, or zero if the script is still running.

Server API

In order for a Charm++ object (chare, array, node, or nodegroup) to receive python requests, it is necessary to define it as python-compliant. This is done through the keyword `python` placed in square brackets before the object name in the .ci file. Some examples follow:

```
mainchare [python] main {...}
array [1D] [python] myArray {...}
group [python] myGroup {...}
```

In order to register a newly created object to receive Python scripts, the method `registerPython` of the proxy should be called. As an example, the following code creates a 10 element array `myArray`, and then registers it to receive scripts directed to “pycode”. The argument of `registerPython` is the string that CCS will use to address the Python scripting capability of the object.

```
Cproxy_myArray localVar = CProxy_myArray::ckNew(10);
localVar.registerPython("pycode");
```

Server read and write functions

As explained previously in subsection 2.4.2, some functions are automatically made available to the scripting code through the `ck` module. Two of these, **read** and **write** are only available if redefined by the object. The signatures of the two methods to redefine are:

```
PyObject* read(PyObject* where);
void write(PyObject* where, PyObject* what);
```

The read function receives as a parameter an object specifying from where the data will be read, and returns an object with the information required. The write function will receive two parameters: where the data will be written and what data, and will perform the update. All these `PyObject`s are generic, and need to be coherent with the protocol

specified by the application. In order to parse the parameters, and create the value of the read, please refer to the manual [Extending and Embedding the Python Interpreter](#), and in particular to the functions `PyArg_ParseTuple` and `Py_BuildValue`.

Server iterator functions

In order to use the iterative mode as explained in subsection [2.4.2](#), it is necessary to implement two functions which will be called by the system. These two functions have the following signatures:

```
int buildIterator(PyObject*, void*);
int nextIteratorUpdate(PyObject*, PyObject*, void*);
```

The first one is called once before the first execution of the Python code, and receives two parameters. The first is a pointer to an empty `PyObject` to be filled with the data needed by the Python code. In order to manage this object, some utility functions are provided. They are explained in subsection [2.4.2](#).

The second is a void pointer containing information of what the iteration should run over. This parameter may contain any data structure, and an agreement between the client and the user object is necessary. The system treats it as a void pointer since it has no information about what user defined data it contains.

The second function (`nextIteratorUpdate`) has three parameters. The first parameter contains the object to be filled (similar to `buildIterator`), but the second object contains the `PyObject` which was provided for the last iteration, potentially modified by the Python function. Its content can be read with the provided routines, used to retrieve the next logical element in the iterator (with which to update the parameter itself), and possibly update the content of the data inside the Charm++ object. The second parameter is the object returned by the last call to the Python function, and the third parameter is the same data structure passed to `buildIterator`.

Both functions return an integer which will be interpreted by the system as follows:

1

- a new iterator in the first parameter has been provided, and the Python function should be called with it;

0

- there are no more elements to iterate.

Server utility functions

These are inherited when declaring an object as Python-compliant, and therefore they are available inside the object code. All of them accept a `PyObject` pointer where to read/write the data, a string with the name of a field, and one or two values containing the data to be read/written (note that to read the data from the `PyObject`, a pointer needs to be passed). The strings used to identify the fields will be the same strings that the Python script will use to access the data inside the object.

The name of the function identifies the type of Python object stored inside the `PyObject` container (i.e String, Int, Long, Float, Complex), while the parameter of the functions identifies the C++ object type.

```
void pythonSetString(PyObject*, char*, char*);
void pythonSetString(PyObject*, char*, char*, int);
void pythonSetInt(PyObject*, char*, long);
void pythonSetLong(PyObject*, char*, long);
void pythonSetLong(PyObject*, char*, unsigned long);
void pythonSetLong(PyObject*, char*, double);
void pythonSetFloat(PyObject*, char*, double);
void pythonSetComplex(PyObject*, char*, double, double);
```

(continues on next page)

(continued from previous page)

```

void pythonGetString(PyObject*, char*, char**);
void pythonGetInt(PyObject*, char*, long*);
void pythonGetLong(PyObject*, char*, long*);
void pythonGetLong(PyObject*, char*, unsigned long*);
void pythonGetLong(PyObject*, char*, double*);
void pythonGetFloat(PyObject*, char*, double*);
void pythonGetComplex(PyObject*, char*, double*, double*);

```

To handle more complicated structures like Dictionaries, Lists or Tuples, please refer to [Python/C API Reference Manual](#).

High level scripting

When in addition to the definition of the Charm++ object as python, an entry method is also defined as python, this entry method can be accessed directly by a Python script through the *charm* module. For example, the following definition will be accessible with the python call: *result = charm.highMethod(var1, var2, var3)* It can accept any number of parameters (even complex like tuples or dictionaries), and it can return an object as complex as needed.

The method must have the following signature:

```
entry [python] void highMethod(int handle);
```

The parameter is a handle that is passed by the system, and can be used in subsequent calls to return values to the Python code.

The arguments passed by the Python caller can be retrieved using the function:

```
PyObject *pythonGetArg(int handle);
```

which returns a PyObject. This object is a Tuple containing a vector of all parameters. It can be parsed using *PyArg_ParseTuple* to extract the single parameters.

When the Charm++'s entry method terminates (by means of *return* or termination of the function), control is returned to the waiting Python script. Since the python entry methods execute within an user-level thread, it is possible to suspend the entry method while some computation is carried on in Charm++. To start parallel computation, the entry method can send regular messages, as every other threaded entry method (see 2.3.2 for more information on how this can be done using *CkCallbackResumeThread* callbacks). The only difference with other threaded entry methods is that here the callback *CkCallbackPython* must be used instead of *CkCallbackResumeThread*. The more specialized *CkCallbackPython* callback works exactly like the other one, except that it correctly handles Python internal locks.

At the end of the computation, the following special function will return a value to the Python script:

```
void pythonReturn(int handle, PyObject* result);
```

where the second parameter is the Python object representing the returned value. The function *Py_BuildValue* can be used to create this value. This function in itself does not terminate the entry method, but only sets the returning value for Python to read when the entry method terminates.

A characteristic of Python is that in a multithreaded environment (like the one provided in Charm++), the running thread needs to keep a lock to prevent other threads to access any variable. When using high level scripting, and the Python script is suspended for long periods of time while waiting for the Charm++ application to perform the required task, the Python internal locks are automatically released and re-acquired by the *CkCallbackPython* class when it suspends.

2.4.3 Intercepting Messages via Delegation

Delegation is a means by which a library writer can intercept messages sent via a proxy. This is typically used to construct communication libraries. A library creates a special kind of Group called a *DelegationManager*, which receives the messages sent via a delegated proxy.

There are two parts to the delegation interface- a very small client-side interface to enable delegation, and a more complex manager-side interface to handle the resulting redirected messages.

Client Interface

All proxies (Chare, Group, Array, ...) in Charm++ support the following delegation routines.

`void CProxy::ckDelegate(CkGroupID delMgr);` Begin delegating messages sent via this proxy to the given delegation manager. This only affects the proxy it is called on- other proxies for the same object are *not* changed. If the proxy is already delegated, this call changes the delegation manager.

`CkGroupID CProxy::ckDelegatedIdx(void) const;` Get this proxy's current delegation manager.

`void CProxy::ckUndelegate(void);` Stop delegating messages sent via this proxy. This restores the proxy to normal operation.

One use of these routines might be:

```
CkGroupID mgr=somebodyElsesCommLib(...);
CProxy_foo p=...;
p.someEntry1(...); //Sent to foo normally
p.ckDelegate(mgr);
p.someEntry2(...); //Handled by mgr, not foo!
p.someEntry3(...); //Handled by mgr again
p.ckUndelegate();
p.someEntry4(...); //Back to foo
```

The client interface is very simple; but it is often not called by users directly. Often the delegate manager library needs some other initialization, so a more typical use would be:

```
CProxy_foo p=...;
p.someEntry1(...); //Sent to foo normally
startCommLib(p,...); // Calls ckDelegate on proxy
p.someEntry2(...); //Handled by library, not foo!
p.someEntry3(...); //Handled by library again
finishCommLib(p,...); // Calls ckUndelegate on proxy
p.someEntry4(...); //Back to foo
```

Sync entry methods, group and nodegroup multicast messages, and messages for virtual chares that have not yet been created are never delegated. Instead, these kinds of entry methods execute as usual, even if the proxy is delegated.

Manager Interface

A delegation manager is a group which inherits from *CkDelegateMgr* and overrides certain virtual methods. Since *CkDelegateMgr* does not do any communication itself, it need not be mentioned in the .ci file; you can simply declare a group as usual and inherit the C++ implementation from *CkDelegateMgr*.

Your delegation manager will be called by Charm++ any time a proxy delegated to it is used. Since any kind of proxy can be delegated, there are separate virtual methods for delegated Chares, Groups, NodeGroups, and Arrays.

```

class CkDelegateMgr : public Group {
public:
    virtual void ChareSend(int ep, void *m, const CkChareID *c, int onPE);

    virtual void GroupSend(int ep, void *m, int onPE, CkGroupID g);
    virtual void GroupBroadcast(int ep, void *m, CkGroupID g);

    virtual void NodeGroupSend(int ep, void *m, int onNode, CkNodeGroupID g);
    virtual void NodeGroupBroadcast(int ep, void *m, CkNodeGroupID g);

    virtual void ArrayCreate(int ep, void *m, const CkArrayIndex &idx, int onPE, CkArrayID a);
    virtual void ArraySend(int ep, void *m, const CkArrayIndex &idx, CkArrayID a);
    virtual void ArrayBroadcast(int ep, void *m, CkArrayID a);
    virtual void ArraySectionSend(int ep, void *m, CkArrayID a, CkSectionID &s);
};

```

These routines are called on the send side only. They are called after parameter marshalling; but before the messages are packed. The parameters passed in have the following descriptions.

1. **ep** The entry point begin called, passed as an index into the Charm++ entry table. This information is also stored in the message's header; it is duplicated here for convenience.
2. **m** The Charm++ message. This is a pointer to the start of the user data; use the system routine `UsrToEnv` to get the corresponding envelope. The messages are not necessarily packed; be sure to use `CkPackMessage`.
3. **c** The destination `CkChareID`. This information is already stored in the message header.
4. **onPE** The destination processor number. For chare messages, this indicates the processor the chare lives on. For group messages, this indicates the destination processor. For array create messages, this indicates the desired processor.
5. **g** The destination `CkGroupID`. This is also stored in the message header.
6. **onNode** The destination node.
7. **idx** The destination array index. This may be looked up using the `lastKnown` method of the array manager, e.g., using:

```
int lastPE=CProxy_CkArray(a).ckLocalBranch()->lastKnown(idx);
```

8. **s** The destination array section.

The *CkDelegateMgr* superclass implements all these methods; so you only need to implement those you wish to optimize. You can also call the superclass to do the final delivery after you've sent your messages.

2.5 Experimental Features

2.5.1 Control Point Automatic Tuning

Charm++ currently includes an experimental automatic tuning framework that can dynamically adapt a program at runtime to improve its performance. The program provides a set of tunable knobs that are adjusted automatically by the tuning framework. The user program also provides information about the control points so that intelligent tuning choices can be made. This information will be used to steer the program instead of requiring the tuning framework to blindly search the possible program configurations.

Warning: this is still an experimental feature not meant for production applications

Exposing Control Points in a Charm++ Program

The program should include a header file before any of its `*.decl.h` files:

```
#include <controlPoints.h>
```

The control point framework initializes itself, so no changes need to be made at startup in the program.

The program will request the values for each control point on PE 0. Control point values are non-negative integers:

```
my_var = controlPoint("any_name", 5, 10);
my_var2 = controlPoint("another_name", 100, 101);
```

To specify information about the effects of each control point, make calls such as these once on PE 0 before accessing any control point values:

```
ControlPoint::EffectDecrease::Granularity("num_chare_rows");
ControlPoint::EffectDecrease::Granularity("num_chare_cols");
ControlPoint::EffectIncrease::NumMessages("num_chare_rows");
ControlPoint::EffectIncrease::NumMessages("num_chare_cols");
ControlPoint::EffectDecrease::MessageSizes("num_chare_rows");
ControlPoint::EffectDecrease::MessageSizes("num_chare_cols");
ControlPoint::EffectIncrease::Concurrency("num_chare_rows");
ControlPoint::EffectIncrease::Concurrency("num_chare_cols");
ControlPoint::EffectIncrease::NumComputeObjects("num_chare_rows");
ControlPoint::EffectIncrease::NumComputeObjects("num_chare_cols");
```

For a complete list of these functions, see `cp_effects.h` in `charm/include`.

The program, of course, has to adapt its behavior to use these new control point values. There are two ways for the control point values to change over time. The program can request that a new phase (with its own control point values) be used whenever it wants, or the control point framework can automatically advance to a new phase periodically. The structure of the program will be slightly different in these two cases. Sections 2.5.1 and 2.5.1 describe the additional changes to the program that should be made for each case.

Control Point Framework Advances Phases

The program provides a callback to the control point framework in a manner such as this:

```
// Once early on in program, create a callback, and register it
CkCallback cb(CkIndex_Main::granularityChange(NULL), thisProxy);
registerCPChangeCallback(cb, true);
```

In the callback or after the callback has executed, the program should request the new control point values on PE 0, and adapt its behavior appropriately.

Alternatively, the program can specify that it wants to call `gotoNextPhase()` itself when it is ready. Perhaps the program wishes to delay its adaptation for a while. To do this, it specifies `false` as the final parameter to `registerCPChangeCallback` as follows:

```
registerCPChangeCallback(cb, false);
```

Program Advances Phases

```
registerControlPointTiming(duration); // called after each program iteration on PE 0
gotoNextPhase(); // called after some number of iterations on PE 0
// Then request new control point values
```

Linking With The Control Point Framework

The control point tuning framework is now an integral part of the Charm++ runtime system. It does not need to be linked in to an application in any special way. It contains the framework code responsible for recording information about the running program as well as adjust the control point values. The trace module will enable measurements to be gathered including information about utilization, idle time, and memory usage.

Runtime Command Line Arguments

Various following command line arguments will affect the behavior of the program when running with the control point framework. As this is an experimental framework, these are subject to change.

The scheme used for tuning can be selected at runtime by the use of one of the following options:

+CPSchemeRandom	Randomly Select Control Point Values
+CPExhaustiveSearch	Exhaustive Search of Control Point Values
+CPSimulAnneal	Simulated Annealing Search of Control Point Values
+CPCriticalPathPrio	Use Critical Path to adapt Control Point Values
+CPBestKnown	Use BestKnown Timing for Control Point Values
+CPSteering	Use Steering to adjust Control Point Values
+CPMemoryAware	Adjust control points to approach available memory

To intelligently tune or steer an application's performance, performance measurements ought to be used. Some of the schemes above require that memory footprint statistics and utilization statistics be gathered. All measurements are performed by a tracing module that has some overheads, so is not enabled by default. To use any type of measurement based steering scheme, it is necessary to add a runtime command line argument to the user program to enable the tracing module:

```
+CPEnableMeasurements
```

The following flags enable the gathering of the different types of available measurements.

+CPGatherAll	Gather all types of measurements for each phase
+CPGatherMemoryUsage	Gather memory usage after each phase
+CPGatherUtilization	Gather utilization & Idle time after each phase

The control point framework will periodically adapt the control point values. The following command line flag determines the frequency at which the control point framework will attempt to adjust things.

```
+CPSamplePeriod    number The time between Control Point Framework samples (in ↵
↵seconds)
```

The data from one run of the program can be saved and used in subsequent runs. The following command line arguments specify that a file named `controlPointData.txt` should be created or loaded. This file contains measurements for each phase as well as the control point values used in each phase.

+CPSaveData	Save Control Point timings & configurations at completion
+CPLoadData	Load Control Point timings & configurations at startup
+CPDataFilename	Specify the data filename

It might be useful for example, to run once with +CPSimulAnneal +CPSaveData to try to find a good configuration for the program, and then use +CPBestKnown +CPLoadData for all subsequent production runs.

2.5.2 Malleability: Shrink/Expand Number of Processors

This feature enables a Charm++ application to dynamically shrink and expand the number of processors that it is running on during the execution. It internally uses three other features of Charm++: CCS (Converse Client Server) interface, load balancing, and checkpoint restart.

- The CCS interface is used to send and receive the shrink and expand commands. These commands can be internal (i.e. application decides to shrink or expand itself) or external via a client. The runtime listens for the commands on a port specified at launch time.
- The load balancing framework is used to evacuate the tasks before a shrink operation and distribute the load equally over the new set of processors after an expand operation.
- The in-memory checkpoint restart mechanism is used to restart the application with the new processor count quickly and without leaving residual processes behind.

An example program with a CCS client to send shrink/expand commands can be found in `examples/charm++/shrink_expand` in the charm distribution.

To enable shrink expand, Charm++ needs to be built with the `--enable-shrinkexpand` option:

```
$ ./build charm++ netlrts-linux-x86_64 --enable-shrinkexpand
```

An example application launch command needs to include a load balancer, a nodelist file that contains all of the nodes that are going to be used, and a port number to listen the shrink/expand commands:

```
$ ./charmrun +p4 ./jacobi2d 200 20 +balancer GreedyLB ++nodelist ./mynodelistfile_
↪ ++server ++server-port 1234
```

The CCS client to send shrink/expand commands needs to specify the hostname, port number, the old(current) number of processor and the new(future) number of processors:

```
$ ./client <hostname> <port> <oldprocs> <newprocs>
(./client valor 1234 4 8 //This will increase from 4 to 8 processors.)
```

To make a Charm++ application malleable, first, pup routines for all of the constructs in the application need to be written. This includes writing a pup routine for the `mainchare` and marking it migratable:

```
mainchare [migratable] Main { ... }
```

Second, the `AtSync()` and `ResumeFromSync()` functions need to be implemented in the usual way of doing load balancing (See Section 2.2.6 for more info on load balancing). Shrink/expand will happen at the next load balancing step after the receipt of the shrink/expand command.

NOTE: If you want to shrink your application, for example, from two physical nodes to one node where each node has eight cores, then you should have eight entries in the nodelist file for each node, one per processor. Otherwise, the application will shrink in a way that will use four cores from each node, whereas what you likely want is to use eight cores on only one of the physical nodes after shrinking. For example, instead of having a nodelist like this:

```
host a
host b
```

the nodelist should be like this:

```
host a
host a
host a
host a
host a
host a
host a
host a
host a
host b
host b
host b
host b
host b
host b
host b
host b
```

Warning: this is an experimental feature and not supported in all charm builds and applications. Currently, it is tested on `netlrts-{linux/darwin}-x86_64` builds. Support for other Charm++ builds and AMPI applications are under development. It is only tested with `RefineLB` and `GreedyLB` load balancing strategies; use other strategies with caution.

2.6 Appendix

2.6.1 Installing Charm++

Charm++ can be installed manually from the source code or using a precompiled binary package. We also provide a source code package for the *Spack* package manager. Building from the source code provides more flexibility, since one can choose the options as desired. However, a precompiled binary may be slightly easier to get running.

Manual installation

Downloading Charm++

Charm++ can be downloaded using one of the following methods:

- From the Charm++ website - The current stable version (source code and binaries) can be downloaded from our website at <https://charm.cs.illinois.edu/software>.
- From source archive - The latest development version of Charm++ can be downloaded from our source archive using `git clone https://github.com/UIUC-PPL/charm`.

If you download the source code from the website, you will have to unpack it using a tool capable of extracting gzip'd tar files, such as `tar` (on Unix) or `WinZIP` (under Windows). Charm++ will be extracted to a directory called `charm`.

Installation

A typical prototype command for building Charm++ from the source code is:


```
$ ./build <TARGET> <TARGET ARCHITECTURE> [OPTIONS]
```

where,

TARGET is the framework one wants to build such as *charm++* or *AMPI*.

TARGET ARCHITECTURE is the machine architecture one wants to build for such as *netlrts-linux-x86_64*, *pamirlts-bluegeneq* etc.

OPTIONS are additional options to the build process, e.g. *smp* is used to build a shared memory version, *-j8* is given to build in parallel etc.

Note: Starting from version 7.0, Charm++ uses the CMake-based build system when building with the `./build` command. To use the old configure-based build system, you can build with the `./buildold` command with the same options. We intend to remove the old build system in Charm++ 7.1.

In Table 2, a list of build commands is provided for some of the commonly used systems. Note that, in general, options such as *smp*, `--with-production`, compiler specifiers etc can be used with all targets. It is advisable to build with `--with-production` to obtain the best performance. If one desires to perform trace collection (for Projections), `--enable-tracing --enable-tracing-commthread` should also be passed to the build command.

Details on all the available alternatives for each of the above mentioned parameters can be found by invoking `./build --help`. One can also go through the build process in an interactive manner. Run `./build`, and it will be followed by a few queries to select appropriate choices for the build one wants to perform.

2: Build command for some common cases

Machine	Build command
Multicore (single node, shared memory) 64 bit Linux	<code>./build charm++ multicore-linux-x86_64 --with-production -j8</code>
Net with 64 bit Linux	<code>./build charm++ netlrts-linux-x86_64 --with-production -j8</code>
Net with 64 bit Linux (intel compilers)	<code>./build charm++ netlrts-linux-x86_64 icc --with-production -j8</code>
Net with 64 bit Linux (shared memory)	<code>./build charm++ netlrts-linux-x86_64 smp --with-production -j8</code>
Net with 64 bit Linux (checkpoint restart based fault tolerance)	<code>./build charm++ netlrts-linux-x86_64 syncft --with-production -j8</code>
Net with 32 bit Linux	<code>./build charm++ netlrts-linux-i386 --with-production -j8</code>
MPI with 64 bit Linux	<code>./build charm++ mpi-linux-x86_64 --with-production -j8</code>
MPI with 64 bit Linux (shared memory)	<code>./build charm++ mpi-linux-x86_64 smp --with-production -j8</code>
MPI with 64 bit Linux (mpicxx wrappers)	<code>./build charm++ mpi-linux-x86_64 mpicxx --with-production -j8</code>
IBVERBS with 64 bit Linux	<code>./build charm++ verbs-linux-x86_64 --with-production -j8</code>
OFI with 64 bit Linux	<code>./build charm++ ofi-linux-x86_64 --with-production -j8</code>
UCX with 64 bit Linux	<code>./build charm++ ucx-linux-x86_64 --with-production -j8</code>
UCX with 64 bit Linux (AArch64)	<code>./build charm++ ucx-linux-arm8 --with-production -j8</code>
Net with 64 bit Windows	<code>./build charm++ netlrts-win-x86_64 --with-production -j8</code>
MPI with 64 bit Windows	<code>./build charm++ mpi-win-x86_64 --with-production -j8</code>
Net with 64 bit macOS (x86_64)	<code>./build charm++ netlrts-darwin-x86_64 --with-production -j8</code>
Net with 64 bit macOS (ARM64)	<code>./build charm++ netlrts-darwin-arm8 --with-production -j8</code>
Blue Gene/Q (bgclang compilers)	<code>./build charm++ pami-bluegeneq --with-production -j8</code>
Blue Gene/Q (bgclang compilers)	<code>./build charm++ pamilrts-bluegeneq --with-production -j8</code>
Cray XE6	<code>./build charm++ gni-crayxe --with-production -j8</code>
Cray XK7	<code>./build charm++ gni-crayxe-cuda --with-production -j8</code>
Cray XC40	<code>./build charm++ gni-crayxc --with-production -j8</code>
Cray Shasta	<code>./build charm++ ofi-crayshasta --with-production -j8</code>

As mentioned earlier, one can also build Charm++ using the precompiled binary in a manner similar to what is used for installing any common software.

When a Charm++ build folder has already been generated, it is possible to perform incremental rebuilds by in-

voking `make` from the `tmp` folder inside it. For example, with a `netlrts-linux-x86_64` build, the path would be `netlrts-linux-x86_64/tmp`. On Linux and macOS, the `tmp` symlink in the top-level charm directory also points to the `tmp` directory of the most recent build.

Alternatively, CMake can be used directly for configuring and building Charm++. You can use `cmake-gui` or `ccmake` for an overview of available options. Note that some are only effective when passed with `-D` from the command line while configuring from a blank slate. To build with all defaults, `cmake .` is sufficient, though invoking CMake from a separate location (ex: `mkdir mybuild && cd mybuild && cmake .`) is recommended. Please see Section 2.6.1 for building Charm++ directly with CMake.

Installation through the Spack package manager

Charm++ can also be installed through the Spack package manager (<https://spack.io/>).

A basic command to install Charm++ through Spack is the following:

```
$ spack install charmpp
```

By default, the `netlrts` network backend with SMP support is built. You can specify other backends by providing the `backend` parameter to `spack`. It is also possible to specify other options, as listed in Section 2.6.1, by adding them to the Spack command prepended by a `+`. For example, to build the MPI version of Charm++ with the integrated OpenMP support, you can use the following command:

```
$ spack install charmpp backend=mpi +omp
```

To disable an option, prepend it with a `~`. For example, to build Charm++ with SMP support disabled, you can use the following command:

```
$ spack install charmpp ~smp
```

By default, the newest released version of Charm++ is built. You can select another version with the `@` option (for example, `@6.8.1`). To build the current git version of Charm++, specify the `@develop` version:

```
$ spack install charmpp@develop
```

Installation with CMake

Charm++ can be installed directly with the CMake tool, version 3.4 or newer, without using the `./build` command.

After downloading and unpacking Charm++, it can be installed in the following way:

```
$ cd charm
$ mkdir build-cmake
$ cd build-cmake
$ cmake ..
$ make -j4
```

By default, CMake builds the `netlrts` version. Other configuration options can be specified in the `cmake` command above. For example, to build Charm++ and AMPI on top of the MPI layer with SMP, the following command can be used:

```
$ cmake .. -DNETWORK=mpi -DSMP=on -DTARGET=AMPI
```

Charm++ installation directories

The main directories in a Charm++ installation are:

charm/bin Executables, such as `charmcc` and `charmrun`, used by Charm++.

charm/doc Documentation for Charm++, such as this document. Distributed as reStructuredText (RST or ReST) source code; HTML and PDF versions can be built or downloaded from our web site.

charm/include The Charm++ C++ and Fortran user include files (.h).

charm/lib The static libraries (.a) that comprise Charm++.

charm/lib_so The shared libraries (.so/.dylib) that comprise Charm++, if Charm++ is compiled with the `-build-shared` option.

charm/examples Example Charm++ programs.

charm/src Source code for Charm++ itself.

charm/tmp Directory where Charm++ is built.

charm/tests Test Charm++ programs used by autobuild.

Reducing disk usage

The `charm` directory contains a collection of example-programs and test-programs. These may be deleted with no other effects. You may also `strip` all the binaries in `charm/bin`.

Installation for Specific Builds

UCX

UCX stands for Unified Communication X and is a high performance communication library that can be used as a backend networking layer for Charm++ builds on supported transports like Mellanox Infiniband, Intel Omni-Path, Cray GNI, TCP/IP, etc.

In order to install Charm++ with the UCX backend, UCX or HPC-X modules are required in your environment. In case UCX and HPC-X are not available in your environment, you can build UCX from scratch using the following steps:

```
$ git clone https://github.com/openucx/ucx.git
$ cd ucx
$ ./autogen.sh
$ ./contrib/configure-release --prefix=$HOME/ucx/build
$ make -j8
$ make install
```

After installing UCX, there are several supported process management interfaces (PMI) that can be specified as options in order to build Charm++ with UCX. These include Simple PMI, Slurm PMI, Slurm PMI 2, and PMIx (included in OpenMPI or OpenPMIx). Currently, in order to use PMIx for process management, it is required to have either OpenMPI or OpenPMIx installed on the system. Additionally, in order to use the other supported process management interfaces, it is required to have a non-OpenMPI based MPI implementation installed on the system (e.g. Intel MPI, MVAPICH, MPICH, etc.).

It should be noted that the **PMIx** version is the *most stable* version of using the UCX backend. We're in the process of debugging some recent issues with Simple PMI, Slurm PMI and Slurm PMI2.

The following section shows examples of build commands that can be used to build targets with the UCX backend using different process management interfaces. It should be noted that unless UCX is installed in a system directory, in order for Charm++ to find the UCX installation, it is required to pass the UCX build directory as `--basedir`.

Simple PMI, Slurm PMI and Slurm PMI 2

To build the Charm++ target with Simple PMI, do not specify any additional option as shown in the build command.

```
$ ./build charm++ ucx-linux-x86_64 --with-production --enable-error-checking --
↳ basedir=$HOME/ucx/build -j16
```

To build the Charm++ target with Slurm PMI, specify `slurmpmi` in the build command.

```
$ ./build charm++ ucx-linux-x86_64 slurmpmi --with-production --enable-error-checking_
↳ --basedir=$HOME/ucx/build -j16
```

Similarly, to build the Charm++ target with Slurm PMI 2, specify `slurmpmi2` in the build command.

```
$ ./build charm++ ucx-linux-x86_64 slurmpmi2 --with-production --enable-error-
↳ checking --basedir=$HOME/ucx/build -j16
```

PMIx (OpenPMIx or as included in OpenMPI)

To build the Charm++ target with PMIx, you can either use OpenPMIx directly or use the version of PMIx included in OpenMPI. Note that PMIx is no longer included in OpenMPI distributions as of v4.0.5, so we recommend building with OpenPMIx instead.

To use OpenPMIx directly, you first need to install libevent (<https://github.com/libevent/libevent>) if it's not available on your system:

```
$ wget https://github.com/libevent/libevent/releases/download/release-2.1.12-stable/
↳ libevent-2.1.12-stable.tar.gz
$ tar -xf libevent-2.1.12-stable.tar.gz
$ cd libevent-2.1.12-stable
$ ./configure --prefix=$HOME/libevent-2.1.12-stable/build
$ make -j
$ make install
```

Then you can download and build OpenPMIx as follows:

```
$ wget https://github.com/openpmix/openpmix/releases/download/v3.1.5/pmix-3.1.5.tar.gz
$ tar -xf pmix-3.1.5.tar.gz
$ cd pmix-3.1.5
$ ./configure --prefix=$HOME/pmix-3.1.5/build --with-libevent=$HOME/libevent-2.1.12-
↳ stable/build
$ make -j
$ make install
```

Finally, Charm++ with the UCX backend can be built with OpenPMIx using the following command (with the OpenPMIx installation passed as an additional `--basedir` argument):

```
$ ./build charm++ ucx-linux-x86_64 openpmix --with-production --enable-error-checking_
↳ --basedir=$HOME/ucx/build --basedir=$HOME/pmix-3.1.5/build -j
```

It should be noted that in the absence of a working launcher such as `jsrun`, an MPI distribution such as OpenMPI may also be required to run Charm++ applications built with the UCX backend and OpenPMIx. Since you no longer need PMIx included with OpenMPI, any version of OpenMPI can be built (including v4.0.5 and later) with your build of OpenPMIx using the `--with-pmix` flag, such as the following:

```
$ wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-4.0.5.tar.gz
$ tar -xf openmpi-4.0.5.tar.gz
$ cd openmpi-4.0.5
$ ./configure --enable-mca-no-build=bt1-uct --with-pmix=$HOME/pmix-3.1.5/build --
  ↳ prefix=$HOME/openmpi-4.0.5/build
$ make -j
$ make install
```

Before executing a Charm++ program, you may need to check that `LD_LIBRARY_PATH` and `PATH` are set to include OpenPMIx (and OpenMPI, if needed).

To use PMIx included with OpenMPI, an OpenMPI implementation with PMIx enabled is required to be installed on your system. As a reminder, PMIx is no longer included in OpenMPI distributions of v4.0.5 or later. In case OpenMPI is not available in your environment, you can build OpenMPI from scratch using the following steps:

```
$ wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-4.0.1.tar.gz
$ tar -xvf openmpi-4.0.1.tar.gz
$ cd openmpi-4.0.1
$ ./configure --enable-install-libpmix --prefix=$HOME/openmpi-4.0.1/build
$ make -j24
$ make install all
```

After installing OpenMPI or using the pre-installed OpenMPI, you can build the Charm++ target with the UCX backend by specifying `ompipmix` in the build command and passing the OpenMPI installation path as `--basedir` (in addition to passing the UCX build directory):

```
$ ./build charm++ ucx-linux-x86_64 ompipmix --with-production --enable-error-checking_
  ↳ --basedir=$HOME/ucx/build --basedir=$HOME/openmpi-4.0.1/build -j16
```

2.6.2 Compiling Charm++ Programs

The `charmcc` program, located in “charm/bin”, standardizes compiling and linking procedures among various machines and operating systems. “charmcc” is a general-purpose tool for compiling and linking, not only restricted to Charm++ programs.

Charmcc can perform the following tasks. The (simplified) syntax for each of these modes is shown. Caution: in reality, one almost always has to add some command-line options in addition to the simplified syntax shown below. The options are described next.

* Compile C	<code>charmcc -o pgm.o pgm.c</code>
* Compile C++	<code>charmcc -o pgm.o pgm.C</code>
* Link	<code>charmcc -o pgm obj1.o obj2.o obj3.o...</code>
* Compile + Link	<code>charmcc -o pgm src1.c src2.ci src3.C</code>
* Create Library	<code>charmcc -o lib.a obj1.o obj2.o obj3.o...</code>
* Translate Charm++ Interface File	<code>charmcc file.ci</code>

Charmcc automatically determines where the charm lib and include directories are — at no point do you have to configure this information. However, the code that finds the lib and include directories can be confused if you remove charmcc from its normal directory, or rearrange the directory tree. Thus, the files in the charm/bin, charm/include, and charm/lib directories must be left where they are relative to each other.

The following command-line options are available to users of charmcc:

-o *output-file*: Output file name. Note: charmcc only ever produces one output file at a time. Because of this, you cannot compile multiple source files at once, unless you then link or archive them into a single output-file. If exactly one source-file is specified, then an output file will be selected by default using the obvious rule (e.g., if

the input file if `pgm.c`, the output file is `pgm.o`). If multiple input files are specified, you must manually specify the name of the output file, which must be a library or executable.

- c:** Ignored. There for compatibility with `cc`.
- Dsymbol [=value]:** Defines preprocessor variables from the command line at compile time.
- I:** Add a directory to the search path for preprocessor include files.
- g:** Causes compiled files to include debugging information.
- L*:** Add a directory to the search path for libraries selected by the `-l` command.
- l*:** Specifies libraries to link in.
- module m1[,m2[,...]]** Specifies additional Charm++ modules to link in. Similar to `-l`, but also registers Charm++ parallel objects. See the library's documentation for whether to use `-l` or `-module`.
- optimize:** Causes files to be compiled with maximum optimization.
- no-optimize:** If this follows `-optimize` on the command line, it turns optimization back off. This can be used to override options specified in makefiles.
- production:** Enable architecture-specific production-mode features. For instance, use available hardware features more aggressively. It's probably a bad idea to build some objects with this, and others without.
- s:** Strip the executable of debugging symbols. Only meaningful when producing an executable.
- verbose:** All commands executed by `charmcc` are echoed to `stdout`.
- seq:** Indicates that we're compiling sequential code. On parallel machines with front ends, this option also means that the code is for the front end. This option is only valid with C and C++ files.
- use-fastest-cc:** Some environments provide more than one C compiler (`cc` and `gcc`, for example). Usually, `charmcc` prefers the less buggy of the two. This option causes `charmcc` to switch to the most aggressive compiler, regardless of whether it's buggy or not.
- use-reliable-cc:** Some environments provide more than one C compiler (`cc` and `gcc`, for example). Usually, `charmcc` prefers the less buggy of the two, but not always. This option causes `charmcc` to switch to the most reliable compiler, regardless of whether it produces slow code or not.
- language {converse|charm++|mpi|fem|f90charm}:** When linking with `charmcc`, one must specify the "language". This is just a way to help `charmcc` include the right libraries. Pick the "language" according to this table:
 - Charm++ if your program includes Charm++, C++, and C.
 - Converse if your program includes C or C++.
 - f90charm if your program includes f90 Charm interface.
- balance seed load-balance-strategy:** When linking any Converse program (including any Charm++ or `sdag` program), one must include a seed load-balancing library. There are currently three to choose from: `rand`, `test`, and `neighbor` are supported. Default is `-balance rand`.

When linking with `neighbor` seed load balancer, one can also specify a virtual topology for constructing neighbors during run-time using `+LBTopo topo`, where *topo* can be one of (a) ring, (b) mesh2d, (c) mesh3d and (d) graph. The default is mesh2d.
- tracemode tracing-mode:** Selects the desired degree of tracing for Charm++ programs. See the Charm++ manual and the Projections manuals for more information. Currently supported modes are `none`, `summary`, and `projections`. Default is `-tracemode none`.
- memory memory-mode:** Selects the implementation of `malloc` and `free` to use. Select a memory mode from the table below.

- **os** Use the operating system's standard memory routines.
- **gnu** Use a set of GNU memory routines.
- **paranoid** Use an error-checking set of routines. These routines will detect common mistakes such as buffer overruns, underruns, double-deletes, and use-after-delete. The extra checks slow down programs, so this version should not be used in production code.
- **leak** Use a special set of memory routines and annotation functions for detecting memory leaks. Call `CmiMemoryMark()` to mark allocated memory as OK, do work which you suspect is leaking memory, then call `CmiMemorySweep(const char *tag)` to check.
- **verbose** Use a tracing set of memory routines. Every memory-related call results in a line printed to standard out.
- **default** Use the default, which depends on the version of Charm++.

-c++ C++ compiler: Forces the specified C++ compiler to be used.

-cc C-compiler: Forces the specified C compiler to be used.

-cp copy-file: Creates a copy of the output file in *copy-file*.

-cpp-option options: Options passed to the C pre-processor.

-ld linker: Use this option only when compiling programs that do not include C++ modules. Forces `charmcc` to use the specified linker.

-ld++ linker: Use this option only when compiling programs that include C++ modules. Forces `charmcc` to use the specified linker.

-ld++-option options: Options passed to the linker for `-language charm++`.

-ld-option options: Options passed to the linker for `-language converse`.

-ldro-option options: Options passes to the linker when linking `.o` files.

Other options that are not listed here will be passed directly to the underlying compiler and/or linker.

2.6.3 Running Charm++ Programs

Launching Programs with `charmrun`

When compiling Charm++ programs, the `charmcc` linker produces both an executable file and an utility called `charmrun`, which is used to load the executable onto the parallel machine.

To run a Charm++ program named “pgm” on four processors, type:

```
$ charmrun pgm +p4
```

Execution on platforms which use platform specific launchers, (i.e., **aprun**, **ibrun**), can proceed without `charmrun`, or `charmrun` can be used in coordination with those launchers via the `++mpiexec` (see Section 2.6.3) parameter.

Programs built using the network version of Charm++ can be run alone, without `charmrun`. This restricts you to using the processors on the local machine, but it is convenient and often useful for debugging. For example, a Charm++ program can be run on one processor in the debugger using:

```
$ gdb pgm
```

If the program needs some environment variables to be set for its execution on compute nodes (such as library paths), they can be set in `.charmrunrc` under home directory. `charmrun` will run that shell script before running the executable.

Charmrun normally limits the number of status messages it prints to a minimal level to avoid flooding the terminal with unimportant information. However, if you encounter trouble launching a job, try using the `++verbose` option to help diagnose the issue. (See the `++quiet` option to suppress output entirely.)

Parameters that function as boolean flags within Charmrun (taking no other parameters) can be prefixed with “no-” to negate their effect. For example, `++no-scalable-start`.

Note: When running on OFI platforms such as Cray Shasta, the OFI runtime parameter `+ofi_runtime_tcp` may be required. By default, the exchange of EP names at startup is done via both PMI and OFI. With this flag, it is only done via PMI.

Command Line Options

A Charm++ program accepts the following command line options:

+auto-provision Automatically determine the number of worker threads to launch in order to fully subscribe the machine running the program.

+autoProvision Same as above.

+oneWthPerHost Launch one worker thread per compute host. By the definition of standalone mode, this always results in exactly one worker thread.

+oneWthPerSocket Launch one worker thread per CPU socket.

+oneWthPerCore Launch one worker thread per CPU core.

+oneWthPerPU Launch one worker thread per CPU processing unit, i.e. hardware thread.

+pN Explicitly request exactly N worker threads. The default is 1.

+ss Print summary statistics about chare creation. This option prints the total number of chare creation requests, and the total number of chare creation requests processed across all processors.

+cs Print statistics about the number of create chare messages requested and processed, the number of messages for chares requested and processed, and the number of messages for branch office chares requested and processed, on a per processor basis. Note that the number of messages created and processed for a particular type of message on a given node may not be the same, since a message may be processed by a different processor from the one originating the request.

user_options Options that are to be interpreted by the user program may be included mixed with the system options. However, `user_options` cannot start with `+`. The `user_options` will be passed as arguments to the user program via the usual `argc/argv` construct to the main entry point of the main chare. Charm++ system options will not appear in `argc/argv`.

Additional Network Options

The following `++` command line options are available in the network version.

First, commands related to subscription of computing resources:

++auto-provision Automatically determine the number of processes and threads to launch in order to fully subscribe the available resources.

++autoProvision Same as above.

++processPerHost N Launch N processes per compute host.

++processPerSocket N Launch N processes per CPU socket.

++processPerCore N Launch N processes per CPU core.

++processPerPU N Launch N processes per CPU processing unit, i.e. hardware thread.

The above options should allow sufficient control over process provisioning for most users. If you need more control, the following advanced options are available:

++n N Run the program with N processes. Functionally identical to `+p` in non-SMP mode (see below). The default is 1.

++p N Total number of processing elements to create. In SMP mode, this refers to worker threads (where $n * ppn = p$), otherwise it refers to processes ($n = p$). The default is 1. Use of `++p` is discouraged in favor of `++processPer*` (and `++oneWithPer*` in SMP mode) where desirable, or `++n` (and `++ppn`) otherwise.

The remaining options cover details of process launch and connectivity:

++local Run charm program only on local machines. No remote shell invocation is needed in this case. It starts node programs right on your local machine. This could be useful if you just want to run small program on only one machine, for example, your laptop.

++mpiexec Use the cluster's `mpiexec` job launcher instead of the built in `ssh` method.

This will pass `-n $P` to indicate how many processes to launch. If `-n $P` is not required because the number of processes to launch is determined via queueing system environment variables then use `++mpiexec-no-n` rather than `++mpiexec`. An executable named something other than `mpiexec` can be used with the additional argument `++remote-shell runmpi`, with `'runmpi'` replaced by the necessary name.

To pass additional arguments to `mpiexec`, specify `++remote-shell` and list them as part of the value after the executable name as follows:

```
$ ./charmrun ++mpiexec ++remote-shell "mpiexec --YourArgumentsHere" ./pgm
```

Use of this option can potentially provide a few benefits:

- Faster startup compared to the SSH approach `charmrun` would otherwise use
- No need to generate a `nodelist` file
- Multi-node job startup on clusters that do not allow connections from the head/login nodes to the compute nodes

At present, this option depends on the environment variables for some common MPI implementations. It supports OpenMPI (`OMPI_COMM_WORLD_RANK` and `OMPI_COMM_WORLD_SIZE`), M(VA)PICH (`MPIRUN_RANK` and `MPIRUN_NPROCS` or `PMI_RANK` and `PMI_SIZE`), and IBM POE (`MP_CHILD` and `MP_PROCS`).

++debug Run each node under `gdb` in an `xterm` window, prompting the user to begin execution.

++debug-no-pause Run each node under `gdb` in an `xterm` window immediately (i.e. without prompting the user to begin execution).

If using one of the `++debug` or `++debug-no-pause` options, the user must ensure the following:

1. The `DISPLAY` environment variable points to your terminal. SSH's X11 forwarding does not work properly with Charm++.
2. The nodes must be authorized to create windows on the host machine (see man pages for `xhost` and `xauth`).
3. `xterm`, `xdpyinfo`, and `gdb` must be in the user's path.
4. The path must be set in the `.cshrc` file, not the `.login` file, because `ssh` does not run the `.login` file.

++scalable-start Scalable start, or SMP-aware startup. It is useful for scalable process launch on multi-core systems since it creates only one ssh session per node and spawns all clients from that ssh session. This is the default startup strategy and the option is retained for backward compatibility.

++batch Ssh a set of node programs at a time, avoiding overloading Charmrun pe. In this strategy, the nodes assigned to a charmrun are divided into sets of fixed size. Charmrun performs ssh to the nodes in the current set, waits for the clients to connect back and then performs ssh on the next set. We call the number of nodes in one ssh set as batch size.

++maxssh Maximum number of ssh's to run at a time. For backwards compatibility, this option is also available as ++maxrsh.

++odelist File containing list of nodes.

++numHosts Number of nodes from the oodelist to use. If the value requested is larger than the number of nodes found, Charmrun will error out.

++help Print help messages

++runscript Script to run node-program with. The specified run script is invoked with the node program and parameter. For example:

```
$ ./charmrun +p4 ./pgm 100 2 3 ++runscript ./set_env_script
```

In this case, the set_env_script is invoked on each node before launching pgm.

++xterm Which xterm to use

++in-xterm Run each node in an xterm window

++display X Display for xterm

++debugger Which debugger to use

++remote-shell Which remote shell to use

++useip Use IP address provided for charmrun IP

++usehostname Send nodes our symbolic hostname instead of IP address

++server-auth CCS Authentication file

++server-port Port to listen for CCS requests

++server Enable client-server (CCS) mode

++nodegroup Which group of nodes to use

++verbose Print diagnostic messages

++quiet Suppress runtime output during startup and shutdown

++timeout Seconds to wait per host connection

++timelimit Seconds to wait for program to complete

SMP Options

SMP mode in Charm++ spawns one OS process per logical node. Within this process there are two types of threads:

1. Worker Threads that have objects mapped to them and execute entry methods
2. Communication Thread that sends and receives data (depending on the network layer)

Charm++ always spawns one communication thread per process when using SMP mode and as many worker threads as the user specifies (see the options below). In general, the worker threads produce messages and hand them to the communication thread, which receives messages and schedules them on worker threads.

To use SMP mode in Charm++, build charm with the `smp` option, e.g.:

```
$ ./build charm++ netlrts-linux-x86_64 smp
```

There are various trade-offs associated with SMP mode. For instance, when using SMP mode there is no waiting to receive messages due to long running entry methods. There is also no time spent in sending messages by the worker threads and memory is limited by the node instead of per core. In SMP mode, intra-node messages use simple pointer passing, which bypasses the overhead associated with the network and extraneous copies. Another benefit is that the runtime will not pollute the caches of worker threads with communication-related data in SMP mode.

However, there are also some drawbacks associated with using SMP mode. First and foremost, you sacrifice one core to the communication thread. This is not ideal for compute bound applications. Additionally, this communication thread may become a serialization bottleneck in applications with large amounts of communication. Keep these trade-offs in mind when evaluating whether to use SMP mode for your application or deciding how many processes to launch per physical node when using SMP mode. Finally, any library code the application may call needs to be thread-safe.

Charm++ provides the following options to control the number of worker threads spawned and the placement of both worker and communication threads:

++oneWthPerHost Launch one worker thread per compute host.

++oneWthPerSocket Launch one worker thread per CPU socket.

++oneWthPerCore Launch one worker thread per CPU core.

++oneWthPerPU Launch one worker thread per CPU processing unit, i.e. hardware thread.

The above options (and `++auto-provision`) should allow sufficient control over thread provisioning for most users. If you need more precise control over thread count and placement, the following options are available:

++ppn N Number of PEs (or worker threads) per logical node (OS process). This option should be specified even when using platform specific launchers (e.g., `aprun`, `ibrun`).

+pemap L[-U[:S[:R]]+O] [, ...] Bind the execution threads to the sequence of cores described by the arguments using the operating system's CPU affinity functions. Can be used outside SMP mode.

A single number identifies a particular core. Two numbers separated by a dash identify an inclusive range (*lower bound* and *upper bound*). If they are followed by a colon and another number (a *stride*), that range will be stepped through in increments of the additional number. Within each stride, a dot followed by a *run* will indicate how many cores to use from that starting point. A plus represents the offset to the previous core number. Multiple `+offset` flags are supported, e.g., `0-7+8+16` equals `0,8,16,1,9,17`.

For example, the sequence `0-8:2,16,20-24` includes cores 0, 2, 4, 6, 8, 16, 20, 21, 22, 23, 24. On a 4-way quad-core system, if one wanted to use 3 cores from each socket, one could write this as `0-15:4.3`. `++ppn 10 +pemap 0-11:6.5+12` equals `++ppn 10 +pemap 0,12,1,13,2,14,3,15,4,16,6,18,7,19,8,20,9,21,10,22`

By default, this option accepts PU indices assigned by the OS. The user might want to instead provide logical PU indices used by `hwloc` (see [link <https://www.openmpi.org/projects/hwloc/doc/v2.1.0/a00342.php#faq_indexes>](https://www.openmpi.org/projects/hwloc/doc/v2.1.0/a00342.php#faq_indexes) for details). To do this, prepend the sequence with an alphabet L (case-insensitive). For instance, `+pemap L0-3` will instruct the runtime to bind threads to PUs with logical indices 0-3.

+commmap p[,q,...] Bind communication threads to the listed cores, one per process.

To run applications in SMP mode, we generally recommend using one logical node per socket or NUMA domain. `++ppn` will spawn N threads in addition to 1 thread spawned by the runtime for the communication threads, so the

total number of threads will be $N+1$ per node. Consequently, you should map both the worker and communication threads to separate cores. Depending on your system and application, it may be necessary to spawn one thread less than the number of cores in order to leave one free for the OS to run on. An example run command might look like:

```
$ ./charmrun ++ppn 3 +p6 +pemap 1-3,5-7 +commap 0,4 ./app <args>
```

This will create two logical nodes/OS processes ($2 = 6$ PEs/3 PEs per node), each with three worker threads/PEs ($++ppn\ 3$). The worker threads/PEs will be mapped thusly: PE 0 to core 1, PE 1 to core 2, PE 2 to core 3 and PE 4 to core 5, PE 5 to core 6, and PE 6 to core 7. PEs/worker threads 0-2 compromise the first logical node and 3-5 are the second logical node. Additionally, the communication threads will be mapped to core 0, for the communication thread of the first logical node, and to core 4, for the communication thread of the second logical node.

Please keep in mind that `+p` always specifies the total number of PEs created by Charm++, regardless of mode (the same number as returned by `CkNumPes()`). The `+p` option does not include the communication thread, there will always be exactly one of those per logical node.

Multicore Options

On multicore platforms, operating systems (by default) are free to move processes and threads among cores to balance load. This however sometimes can degrade the performance of Charm++ applications due to the extra overhead of moving processes and threads, especially for Charm++ applications that already implement their own dynamic load balancing.

Charm++ provides the following runtime options to set the processor affinity automatically so that processes or threads no longer move. When cpu affinity is supported by an operating system (tested at Charm++ configuration time), the same runtime options can be used for all flavors of Charm++ versions including network and MPI versions, smp and non-smp versions.

+setcpuaaffinity Set cpu affinity automatically for processes (when Charm++ is based on non-smp versions) or threads (when smp). This option is recommended, as it prevents the OS from unnecessarily moving processes/threads around the processors of a physical node.

+excludecore <core #> Do not set cpu affinity for the given core number. One can use this option multiple times to provide a list of core numbers to avoid.

IO buffering options

There may be circumstances where a Charm++ application may want to take or relinquish control of stdout buffer flushing. Most systems default to giving the Charm++ runtime control over stdout but a few default to giving the application that control. The user can override these system defaults with the following runtime options:

+io_flush_user User (application) controls stdout flushing

+io_flush_system The Charm++ runtime controls flushing

Nodelist file

For network of workstations, the list of machines to run the program can be specified in a file. Without a nodelist file, Charm++ runs the program only on the local machine.

The format of this file allows you to define groups of machines, giving each group a name. Each line of the nodes file is a command. The most important command is:

```
host <hostname> <qualifiers>
```

which specifies a host. The other commands are qualifiers: they modify the properties of all hosts that follow them. The qualifiers are:

`group <groupname>` - subsequent hosts are members of specified group
`login <login>` - subsequent hosts use the specified login
`shell <shell>` - subsequent hosts use the specified remote shell
`setup <cmd>` - subsequent hosts should execute cmd
`pathfix <dir1> <dir2>` - subsequent hosts should replace dir1 with dir2 in the program path
`cpus <n>` - subsequent hosts should use N light-weight processes
`speed <s>` - subsequent hosts have relative speed rating
`ext <extn>` - subsequent hosts should append extn to the pgm name

Note: By default, `charmrun` uses a remote shell “ssh” to spawn node processes on the remote hosts. The `shell` qualifier can be used to override it with say, “rsh”. One can set the `CONV_RSH` environment variable or use `charmrun` option `++remote-shell` to override the default remote shell for all hosts with unspecified `shell` qualifier.

All qualifiers accept “*” as an argument, this resets the modifier to its default value. Note that currently, the `passwd`, `cpus`, and `speed` factors are ignored. Inline qualifiers are also allowed:

```
host beauty ++cpus 2 ++shell ssh
```

Except for “group”, every other qualifier can be inlined, with the restriction that if the “setup” qualifier is inlined, it should be the last qualifier on the “host” or “group” statement line.

Here is a simple nodes file:

```
group kale-sun ++cpus 1
  host charm.cs.illinois.edu ++shell ssh
  host dp.cs.illinois.edu
  host grace.cs.illinois.edu
  host dagger.cs.illinois.edu
group kale-sol
  host beauty.cs.illinois.edu ++cpus 2
group main
  host localhost
```

This defines three groups of machines: group `kale-sun`, group `kale-sol`, and group `main`. The `++nodegroup` option is used to specify which group of machines to use. Note that there is wraparound: if you specify more nodes than there are hosts in the group, it will reuse hosts. Thus,

```
$ charmrun pgm ++nodegroup kale-sun +p6
```

uses hosts (`charm`, `dp`, `grace`, `dagger`, `charm`, `dp`) respectively as nodes (0, 1, 2, 3, 4, 5).

If you don’t specify a `++nodegroup`, the default is `++nodegroup main`. Thus, if one specifies

```
$ charmrun pgm +p4
```

it will use “localhost” four times. “localhost” is a Unix trick; it always find a name for whatever machine you’re on.

Using “ssh”, the user will have to setup password-less login to remote hosts using public key authentication based on a key-pair and adding public keys to “`ssh/authorized_keys`” file. See “ssh” documentation for more information. If “rsh” is used for remote login to the compute nodes, the user is required to set up remote login permissions on all nodes using the “`.rhosts`” file in their home directory.

In a network environment, `charmrun` must be able to locate the directory of the executable. If all workstations share a common file name space this is trivial. If they don't, `charmrun` will attempt to find the executable in a directory with the same path from the **\$HOME** directory. Pathname resolution is performed as follows:

1. The system computes the absolute path of `pgm`.
2. If the absolute path starts with the equivalent of **\$HOME** or the current working directory, the beginning part of the path is replaced with the environment variable **\$HOME** or the current working directory. However, if `++pathfix dir1 dir2` is specified in the nodes file (see above), the part of the path matching `dir1` is replaced with `dir2`.
3. The system tries to locate this program (with modified pathname and appended extension if specified) on all nodes.

Instructions to run Charm++ programs for Specific Builds

UCX

When Charm++ is built with UCX using Simple PMI, Slurm PMI or Slurm PMI2, ensure that an MPI implementation which is not OpenMPI is loaded in the environment. Use the `mpirun` launcher provided by the non-OpenMPI implementation to launch programs. Alternatively, you can also use any system provided launchers.

When Charm++ is built with UCX using PMIx, ensure that an OpenMPI implementation is loaded in the environment. Use the `mpiexec/mpirun` launcher provided with OpenMPI to launch programs.

Note that due to bug #2477, binaries built with UCX cannot be run in standalone mode i.e without a launcher.

2.6.4 Reserved words in `.ci` files

The following words are reserved for the Charm++ interface translator, and cannot appear as variable or entry method names in a `.ci` file:

- `module`
- `mainmodule`
- `chare`
- `mainchare`
- `group`
- `nodegroup`
- `namespace`
- `array`
- `message`
- `conditional`
- `extern`
- `initcall`
- `initnode`
- `initproc`
- `readonly`
- `PUPable`

- pupable
- template
- class
- include
- virtual
- packed
- varsize
- entry
- using
- nocopy
- nocopypost
- nocopydevice
- migratable
- python
- Entry method attributes
 - stacksize
 - threaded
 - createhere
 - createhome
 - sync
 - iget
 - exclusive
 - immediate
 - expedited
 - inline
 - local
 - aggregate
 - nokeep
 - notrace
 - accel (reserved for future/experimental use)
 - readwrite (reserved for future/experimental use)
 - writeonly (reserved for future/experimental use)
 - accelblock (reserved for future/experimental use)
 - memcritical
 - reductiontarget
- Basic C++ types

- int
- short
- long
- char
- float
- double
- unsigned
- void
- const
- SDAG constructs
 - atomic
 - serial
 - when
 - while
 - for
 - forall
 - if
 - else
 - overlap

2.6.5 Performance Tracing for Analysis

Projections is a performance analysis/visualization framework that helps you understand and investigate performance-related problems in the (Charm++) applications. It is a framework with an event tracing component which allows to control the amount of information generated. The tracing has low perturbation on the application. It also has a Java-based visualization and analysis component with various views that help present the performance information in a visually useful manner.

Performance analysis with Projections typically involves two simple steps:

1. Prepare your application by linking with the appropriate trace generation modules and execute it to generate trace data.
2. Using the Java-based tool to visually study various aspects of the performance and locate the performance issues for that application execution.

The Charm++ runtime automatically records pertinent performance data for performance-related events during execution. These events include the start and end of entry method execution, message send from entry methods and scheduler idle time. This means *most* users do not need to manually insert code into their applications in order to generate trace data. In scenarios where special performance information not captured by the runtime is required, an API (see section 2.6.5) is available for user-specific events with some support for visualization by the Java-based tool. If greater control over tracing activities (e.g. dynamically turning instrumentation on and off) is desired, the API also allows users to insert code into their applications for such purposes.

The automatic recording of events by the Projections framework introduces the overhead of an if-statement for each runtime event, even if no performance analysis traces are desired. Developers of Charm++ applications who consider such an overhead to be unacceptable (e.g. for a production application which requires the absolute best performance)

may recompile the Charm++ runtime with the `--with-production` flag, which removes the instrumentation stubs. To enable the instrumentation stubs while retaining the other optimizations enabled by `--with-production`, one may compile Charm++ with both `--with-production` and `--enable-tracing`, which explicitly enables Projections tracing.

To enable performance tracing of your application, users simply need to link the appropriate trace data generation module(s) (also referred to as *tracemode(s)*). (see section 2.6.5)

Enabling Performance Tracing at Link/Run Time

Projections tracing modules dictate the type of performance data, data detail and data format each processor will record. They are also referred to as “tracemodes”. There are currently 2 tracemodes available. Zero or more trace-modes may be specified at link-time. When no tracemodes are specified, no trace data is generated.

Tracemode projections

Link time option: `-tracemode projections`

This tracemode generates files that contain information about all Charm++ events like entry method calls and message packing during the execution of the program. The data will be used by Projections in visualization and analysis.

This tracemode creates a single symbol table file and *p* ASCII log files for *p* processors. The names of the log files will be `NAME.#.log` where `NAME` is the name of your executable and `#` is the processor `#`. The name of the symbol table file is `NAME.sts` where `NAME` is the name of your executable.

This is the main source of data needed by the performance visualizer. Certain tools like timeline will not work without the detail data from this tracemode.

The following is a list of runtime options available under this tracemode:

- `+logsize NUM`: keep only `NUM` log entries in the memory of each processor. The logs are emptied and flushed to disk when filled. (defaults to 1,000,000)
- `+binary-trace`: generate projections log in binary form.
- `+gz-trace`: generate gzip (if available) compressed log files.
- `+no-gz-trace`: generate regular (uncompressed) log files.
- `+notracenested`: a debug option. Does not resume tracing outer entry methods when entry methods are nested (as can happen with `[local]` and `[inline]` calls.
- `+checknested`: a debug option. Checks if events are improperly nested while recorded and issue a warning immediately.
- `+trace-subdirs NUM`: divide the generated log files among `NUM` subdirectories of the trace root, each named `PROGNAME.projdir.K`

Tracemode summary

Link time option: `-tracemode summary`

In this tracemode, execution time across all entry points for each processor is partitioned into a fixed number of equally sized time-interval bins. These bins are globally resized whenever they are all filled in order to accommodate longer execution times while keeping the amount of space used constant.

Additional data like the total number of calls made to each entry point is summarized within each processor.

This tracemode will generate a single symbol table file and p ASCII summary files for p processors. The names of the summary files will be `NAME.#.sum` where `NAME` is the name of your executable and `#` is the processor #. The name of the symbol table file is `NAME.sum.sts` where `NAME` is the name of your executable.

This tracemode can be used to control the amount of output generated in a run. It is typically used in scenarios where a quick look at the overall utilization graph of the application is desired to identify smaller regions of time for more detailed study. Attempting to generate the same graph using the detailed logs of the prior tracemode may be unnecessarily time consuming or impossible.

The following is a list of runtime options available under this tracemode:

- `+bincount NUM`: use `NUM` time-interval bins. The bins are resized and compacted when filled.
- `+binsize TIME`: sets the initial time quantum each bin represents.
- `+version`: set summary version to generate.
- `+sumDetail`: Generates a additional set of files, one per processor, that stores the time spent by each entry method associated with each time-bin. The names of “summary detail” files will be `NAME.#.sumd` where `NAME` is the name of your executable and `#` is the processor #.
- `+sumonly`: Generates a single file that stores a single utilization value per time-bin, averaged across all processors. This file bears the name `NAME.sum` where `NAME` is the name of your executable. This runtime option currently overrides the `+sumDetail` option.

General Runtime Options

The following is a list of runtime options available with the same semantics for all tracemodes:

- `+traceroot DIR`: place all generated files in `DIR`.
- `+traceoff`: trace generation is turned off when the application is started. The user is expected to insert code to turn tracing on at some point in the run.
- `+traceWarn`: By default, warning messages from the framework are not displayed. This option enables warning messages to be printed to screen. However, on large numbers of processors, they can overwhelm the terminal I/O system of the machine and result in unacceptable perturbation of the application.
- `+traceprocessors RANGE`: Only output logfiles for PEs present in the range (i.e. `0-31, 32-999966:1000, 999967-999999` to record every PE on the first 32, only every thousandth for the middle range, and the last 32 for a million processor run).

End-of-run Analysis for Data Reduction

As applications are scaled to thousands or hundreds of thousands of processors, the amount of data generated becomes extremely large and potentially unmanageable by the visualization tool. At the time of documentation, Projections is capable of handling data from 8000+ processors but with somewhat severe tool responsiveness issues. We have developed an approach to mitigate this data size problem with options to trim-off “uninteresting” processors’ data by not writing such data at the end of an application’s execution.

This is currently done through heuristics to pick out interesting extremal (i.e. poorly behaved) processors and at the same time using a k-means clustering to pick out exemplar processors from equivalence classes to form a representative subset of processor data. The analyst is advised to also link in the summary module via `+tracemode summary` and enable the `+sumDetail` option in order to retain some profile data for processors whose data were dropped.

- `+extrema`: enables extremal processor identification analysis at the end of the application’s execution.

- `+numClusters`: determines the number of clusters (equivalence classes) to be used by the k-means clustering algorithm for determining exemplar processors. Analysts should take advantage of their knowledge of natural application decomposition to guess at a good value for this.

This feature is still being developed and refined as part of our research. It would be appreciated if users of this feature could contact the developers if you have input or suggestions.

Controlling Tracing from Within the Program

Selective Tracing

Charm++ allows users to start/stop tracing the execution at certain points in time on the local processor. Users are advised to make these calls on all processors and at well-defined points in the application.

Users may choose to have instrumentation turned off at first (by command line option `+traceoff` - see section 2.6.5) if some period of time in middle of the application's execution is of interest to the user.

Alternatively, users may start the application with instrumentation turned on (default) and turn off tracing for specific sections of the application.

Again, users are advised to be consistent as the `+traceoff` runtime option applies to all processors in the application.

- `void traceBegin()`
Enables the runtime to trace events (including all user events) on the local processor where `traceBegin` is called.
- `void traceEnd()`
Prevents the runtime from tracing events (including all user events) on the local processor where `traceEnd` is called.

Explicit Flushing

By default, when linking with `-tracemode projections`, log files are flushed to disk whenever the number of entries on a processor reaches the logsize limit (see Section 2.6.5). However, this can occur at any time during the execution of the program, potentially causing performance perturbations. To address this, users can explicitly flush to disk using the `traceFlushLog()` function. Note that automatic flushing will still occur if the logsize limit is reached, but sufficiently frequent explicit flushes should prevent that from happening.

- `void traceFlushLog()`
Explicitly flushes the collected logs to disk.

User Events

Projections has the ability to visualize traceable user specified events. User events are usually displayed in the Timeline view as vertical bars above the entry methods. Alternatively the user event can be displayed as a vertical bar that vertically spans the timelines for all processors. Follow these following basic steps for creating user events in a charm++ program:

1. Register an event with an identifying string and either specify or acquire a globally unique event identifier. All user events that are not registered will be displayed in white.
2. Use the event identifier to specify trace points in your code of interest to you.

The functions available are as follows:

- `int traceRegisterUserEvent(char* EventDesc, int EventNum=-1)`

This function registers a user event by associating `EventNum` to `EventDesc`. If `EventNum` is not specified, a globally unique event identifier is obtained from the runtime and returned. The string `EventDesc` must either be a constant string, or it can be a dynamically allocated string that is **NOT** freed by the program. If the `EventDesc` contains a substring `***` then the Projections Timeline tool will draw the event vertically spanning all PE timelines.

`EventNum` has to be the same on all processors. Therefore use one of the following methods to ensure the same value for any PEs generating the user events:

1. Call `traceRegisterUserEvent` on PE 0 in `main::main` without specifying an event number, and store returned event number into a readonly variable.
2. Call `traceRegisterUserEvent` and specify the event number on processor 0. Doing this on other processors would have no effect. Afterwards, the event number can be used in the following user event calls.

Eg. `traceRegisterUserEvent("Time Step Begin", 10);`

Eg. `eventID = traceRegisterUserEvent("Time Step Begin");`

There are two main types of user events, bracketed and non bracketed. Non-bracketed user events mark a specific point in time. Bracketed user events span an arbitrary contiguous time range. Additionally, the user can supply a short user supplied text string that is recorded with the event in the log file. These strings should not contain newline characters, but they may contain simple html formatting tags such as `
`, ``, `<i>`, ``, etc.

The calls for recording user events are the following:

- `void traceUserEvent(int EventNum)`

This function creates a user event that marks a specific point in time.

Eg. `traceUserEvent(10);`

- `void traceBeginUserBracketEvent(int EventNum)`

`void traceEndUserBracketEvent(int EventNum)`

These functions record a user event spanning a time interval. The tracing framework automatically associates the call with the time it was made, so timestamps are not explicitly passed in as they are with `traceUserBracketEvent`.

- `void traceUserBracketEvent(int EventNum, double StartTime, double EndTime)`

This function records a user event spanning a time interval from `StartTime` to `EndTime`. Both `StartTime` and `EndTime` should be obtained from a call to `CmiWallTimer()` at the appropriate point in the program.

Eg.

```
traceRegisterUserEvent("Critical Code", 20); // on PE 0
double critStart = CmiWallTimer(); // start time
// do the critical code
traceUserBracketEvent(20, critStart, CmiWallTimer());
```

- `void traceUserSuppliedData(int data)`

This function records a user specified data value at the current time. This data value can be used to color entry method invocations in Timeline, see [15.2.3](#).

- `void traceUserSuppliedNote(char * note)`

This function records a user specified text string at the current time.

- `void traceUserSuppliedBracketedNote(char *note, int EventNum, double StartTime, double EndTime)`

This function records a user event spanning a time interval from `StartTime` to `EndTime`. Both `StartTime` and `EndTime` should be obtained from a call to `CmiWallTimer()` at the appropriate point in the program.

Additionally, a user supplied text string is recorded, and the `EventNum` is recorded. These events are therefore displayed with colors determined by the `EventNum`, just as those generated with `traceUserBracketEvent` are.

User Stats

Charm++ allows the user to track the progression of any variable or value throughout the program execution. These user specified stats can then be plotted in Projections, either over time or by processor. To enable this feature for Charm++, build Charm++ with the `-enable-tracing` flag.

Follow these steps to track user stats in a Charm++ program:

1. Register a stat with an identifying string and a globally unique integer identifier.
2. Update the value of the stat at points of interest in the code by calling the update stat functions.
3. Compile program with `-tracemode projections` flag.

The functions available are as follows:

- `int traceRegisterUserStat(const char * StatDesc, int StatNum)`

This function is called once near the beginning the of the Charm++ program. `StatDesc` is the identifying string and `StatNum` is the unique integer identifier.

- `void updateStat(int StatNum, double StatValue)`

This function updates the value of a user stat and can be called many times throughout program execution. `StatNum` is the integer identifier corresponding to the desired stat. `StatValue` is the updated value of the user stat.

- `void updateStatPair(int StatNum, double StatValue, double Time)`

This function works similar to `updateStat()`, but also allows the user to store a user specified time for the update. In Projections, the user can then choose which time scale to use: real time, user specified time, or ordered.

Function-level Tracing for Adaptive MPI Applications

Adaptive MPI (AMPI) is an implementation of the MPI interface on top of Charm++. As with standard MPI programs, the appropriate semantic context for performance analysis is captured through the observation of MPI calls within C/C++/Fortran functions. Users can selectively begin and end tracing in AMPI programs using the routines `AMPI_Trace_begin` and `AMPI_Trace_end`.

2.6.6 Debugging

Message Order Race Conditions

While common Charm++ programs are data-race free due to a lack of shared mutable state between threads, it is still possible to observe race conditions resulting from variation in the order that messages are delivered to each chare object. The Charm++ ecosystem offers a variety of ways to attempt to reproduce these often non-deterministic bugs, diagnose their causes, and test fixes.

Randomized Message Queueing

To facilitate debugging of applications and to identify race conditions due to message order, the user can enable a randomized message scheduler queue. Using the build-time configuration option `--enable-randomized-msgq`, the charm message queue will be randomized. Note that a randomized message queue is only available when message priority type is not bit vector. Therefore, the user needs to specify `prio-type` to be a data type long enough to hold the msg priorities in your application for eg: `--with-prio-type=int`.

CharmDebug

The CharmDebug interactive debugging tool can be used to inspect the messages in the scheduling queue of each processing element, and to manipulate the order in which they're delivered. More details on how to use CharmDebug can be found in its manual.

Deterministic Record-Replay

Charm++ supports recording the order in which messages are processed from one run, to deterministically replay the same order in subsequent runs. This can be useful to capture the infrequent undesirable message order cases that cause intermittent failures. Once an impacted run has been recorded, various debugging methods can be more easily brought to bear, without worrying that they will perturb execution to avoid the bug.

Support for record-replay is enabled in common builds of Charm++. Builds with either of the `--with-production` or `--disable-tracing` options disable record-replay support. To record traces, simply run the program with an additional command line-flag `+record`. The generated traces can be repeated with the command-line flag `+replay`. The full range of parallel and sequential debugging techniques are available to apply during deterministic replay.

The traces will work even if the application is modified and recompiled, as long as entry method numbering and send/receive sequences do not change. For instance, it is acceptable to add print statements or assertions to aid in the debugging process.

Memory Access Errors

Using Valgrind

The popular Valgrind memory debugging tool can be used to monitor Charm++ applications in both serial and parallel executions. For single-process runs, it can be used directly:

```
$ valgrind ...valgrind options... ./application_name ...application arguments...
```

When running in parallel, it is helpful to note a few useful adaptations of the above incantation, for various kinds of process launchers:

```
$ ./charmrun +p2 `which valgrind` --log-file=VG.out.%p --trace-children=yes ./
↪ application_name ...application arguments...
$ aprun -n 2 `which valgrind` --log-file=VG.out.%p --trace-children=yes ./application_
↪ name ...application arguments...
```

The first adaptation is to use ``which valgrind`` to obtain a full path to the valgrind binary, since parallel process launchers typically do not search the environment `$PATH` directories for the program to run. The second adaptation is found in the options passed to valgrind. These will make sure that valgrind tracks the spawned application process, and write its output to per-process logs in the file system rather than standard error.

2.6.7 History

The Charm software was developed as a group effort of the Parallel Programming Laboratory at the University of Illinois at Urbana-Champaign. Researchers at the Parallel Programming Laboratory keep Charm++ updated for the new machines, new programming paradigms, and for supporting and simplifying development of emerging applications for parallel processing. The earliest prototype, Chare Kernel(1.0), was developed in the late eighties. It consisted only of basic remote method invocation constructs available as a library. The second prototype, Chare Kernel(2.0), a complete re-write with major design changes. This included C language extensions to denote Chares, messages and asynchronous remote method invocation. Charm(3.0) improved on this syntax, and contained important features such as information sharing abstractions, and chare groups (called Branch Office Chares). Charm(4.0) included Charm++ and was released in fall 1993. Charm++ in its initial version consisted of syntactic changes to C++ and employed a special translator that parsed the entire C++ code while translating the syntactic extensions. Charm(4.5) had a major change that resulted from a significant shift in the research agenda of the Parallel Programming Laboratory. The message-driven runtime system code of the Charm++ was separated from the actual language implementation, resulting in an interoperable parallel runtime system called Converse. The Charm++ runtime system was retargetted on top of Converse, and popular programming paradigms such as MPI and PVM were also implemented on Converse. This allowed interoperability between these paradigms and Charm++. This release also eliminated the full-fledged Charm++ translator by replacing syntactic extensions to C++ with C++ macros, and instead contained a small language and a translator for describing the interfaces of Charm++ entities to the runtime system. This version of Charm++, which, in earlier releases was known as *Interface Translator Charm++*, is the default version of Charm++ now, and hence referred simply as **Charm++**. In early 1999, the runtime system of Charm++ was rewritten in C++. Several new features were added. The interface language underwent significant changes, and the macros that replaced the syntactic extensions in original Charm++, were replaced by natural C++ constructs. Late 1999, and early 2000 reflected several additions to Charm++, when a load balancing framework and migratable objects were added to Charm++.

2.6.8 Acknowledgements

- Aaron Becker
- Abhinav Bhatele
- Abhishek Gupta
- Akhil Langer
- Amit Sharma
- Anshu Arya
- Artem Shvorin
- Arun Singla
- Attila Gursoy
- Bilge Acun
- Chao Huang
- Chao Mei
- Chee Wai Lee
- Cyril Bordage
- David Kunzman
- Dmitriy Ofman
- Edgar Solomonik

- Ehsan Totoni
- Emmanuel Jeannot
- Eric Bohm
- Eric Mikida
- Eric Shook
- Esteban Meneses
- Esteban Pauli
- Evan Ramos
- Filippo Gioachin
- Gengbin Zheng
- Greg Koenig
- Gunavardhan Kakulapati
- Hari Govind
- Harshit Dokania
- Harshitha Menon
- Isaac Dooley
- Jaemin Choi
- Jayant DeSouza
- Jeffrey Wright
- Jim Phillips
- Jonathan Booth
- Jonathan Lifflander
- Joshua Unger
- Josh Yelon
- Juan Galvez
- Justin Szaday
- Kavitha Chandrasekar
- Laxmikant Kale
- Lixia Shi
- Lukasz Wesolowski
- Mani Srinivas Potnuru
- Matthias Diener
- Michael Robson
- Milind Bhandarkar
- Minas Charalambides
- Narain Jagathesan

- Neelam Saboo
- Nihit Desai
- Nikhil Jain
- Nilesch Choudhury
- Nitin Bhat
- Orion Lawlor
- Osman Sarood
- Parthasarathy Ramachandran
- Pathikrit Ghosh
- Phil Miller
- Prateek Jindal
- Pritish Jetley
- Puneet Narula
- Raghavendra Kanakagiri
- Rahul Joshi
- Ralf Gunter
- Ramkumar Vadali
- Ramprasad Venkataraman
- Rashmi Jyothi
- Robert Blake
- Robert Brunner
- Ronak Buch
- Rui Liu
- Ryan Mokos
- Sam White
- Sameer Kumar
- Sameer Paranjpye
- Sanjeev Krishnan
- Sayantan Chakravorty
- Seonmyeong Bak
- Sindhura Bandhakavi
- Tarun Agarwal
- Terry L. Wilmarth
- Theckla Louchios
- Tim Hinrichs
- Timothy Knauff

- Venkatasubrahmanian Narayanan
- Vikas Mehta
- Viraj Paropkari
- Vipul Harsh
- Xiang Ni
- Yanhua Sun
- Yan Shi
- Yogesh Mehta
- Zane Fink
- Zheng Shao

Adaptive MPI (AMPI)

Contents

- *Adaptive MPI (AMPI)*

3.1 Introduction

This manual describes Adaptive MPI (AMPI), which is an implementation of the MPI standard on top of Charm++. AMPI acts as a regular MPI implementation (akin to MPICH, OpenMPI, MVAPICH, etc.) with several built-in extensions that allow MPI developers to take advantage of Charm++'s dynamic runtime system, which provides support for process virtualization, overlap of communication and computation, load balancing, and fault tolerance with zero to minimal changes to existing MPI codes.

In this manual, we first describe the philosophy behind Adaptive MPI, then give a brief introduction to Charm++ and rationale for AMPI. We then describe AMPI in detail. Finally we summarize the changes required for existing MPI codes to run with AMPI. Appendices contain the details of installing AMPI, and building and running AMPI programs.

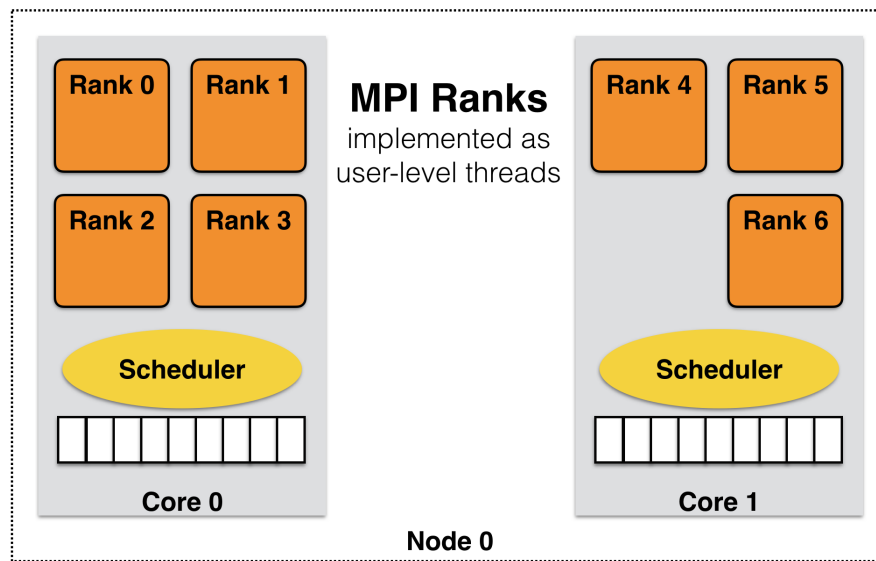
Note: Currently, AMPI supports the MPI-2.2 standard, and the MPI-3.1 standard is under active development, though we already support non-blocking and neighborhood collectives among other MPI-3.1 features.

3.1.1 Overview

Developing parallel Computational Science and Engineering (CSE) applications is a complex task. One has to implement the right physics, develop or choose and code appropriate numerical methods, decide and implement the proper input and output data formats, perform visualizations, and be concerned with correctness and efficiency of the programs. It becomes even more complex for multi-physics coupled simulations, many of which are dynamic and adaptively refined so that load imbalance becomes a major challenge. In addition to imbalance caused by dynamic

program behavior, hardware factors such as latencies, variability, and failures must be tolerated by applications. Our philosophy is to lessen the burden of application developers by providing advanced programming paradigms and versatile runtime systems that can handle many common programming and performance concerns automatically and let application programmers focus on the actual application content.

Many of these concerns can be addressed using the processor virtualization and over-decomposition philosophy of Charm++. Thus, the developer only sees virtual processors and lets the runtime system deal with underlying physical processors. This is implemented in AMPI by mapping MPI ranks to Charm++ user-level threads as illustrated in Figure 3. As an immediate and simple benefit, the programmer can use as many virtual processors (“MPI ranks”) as the problem can be easily decomposed to. For example, suppose the problem domain has $n * 2^n$ parts that can be easily distributed but programming for general number of MPI processes is burdensome, then the developer can have $n * 2^n$ virtual processors on any number of physical ones using AMPI.



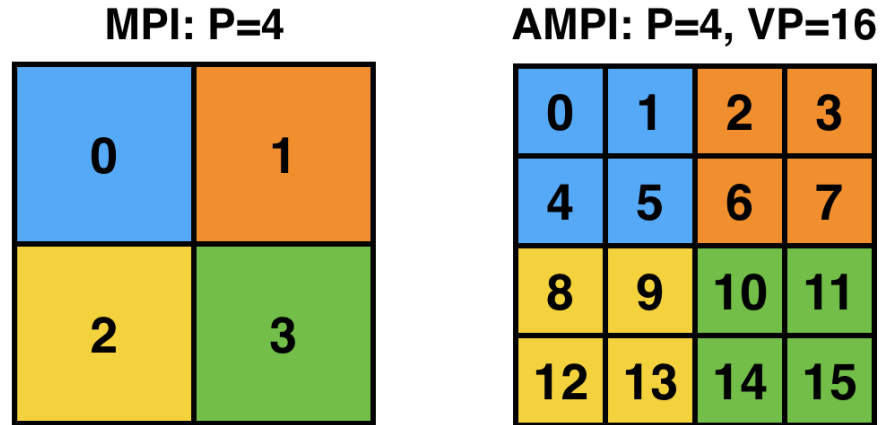
3: MPI ranks are implemented as user-level threads in AMPI rather than Operating System processes.

AMPI’s execution model consists of multiple user-level threads per Processing Element (PE). The Charm++ scheduler coordinates execution of these user-level threads (also called Virtual Processors or VPs) and controls execution. These VPs can also migrate between PEs for the purpose of load balancing or other reasons. The number of VPs per PE specifies the virtualization ratio (degree of over-decomposition). For example, in Figure 3 the virtualization ratio is 3.5 (there are four VPs on PE 0 and three VPs on PE 1). Figure 4 shows how the problem domain can be over-decomposed in AMPI’s VPs as opposed to other MPI implementations.

Another benefit of virtualization is communication and computation overlap, which is automatically realized in AMPI without programming effort. Techniques such as software pipelining require significant programming effort to achieve this goal and improve performance. However, one can use AMPI to have more virtual processors than physical processors to overlap communication and computation. Each time a VP is blocked for communication, the Charm++ scheduler picks the next VP among those that are ready to execute. In this manner, while some of the VPs of a physical processor are waiting for a message to arrive, others can continue their execution. Thus, performance improves without any changes to the application source code.

Another potential benefit is that of better cache utilization. With over-decomposition, a smaller subdomain is accessed by a VP repeatedly in different function calls before getting blocked by communication and switching to another VP. That smaller subdomain may fit into cache if over-decomposition is enough. This concept is illustrated in Figure 3 where each AMPI rank’s subdomain is smaller than the corresponding MPI subdomain and so may fit into cache memory. Thus, there is a potential performance improvement without changing the source code.

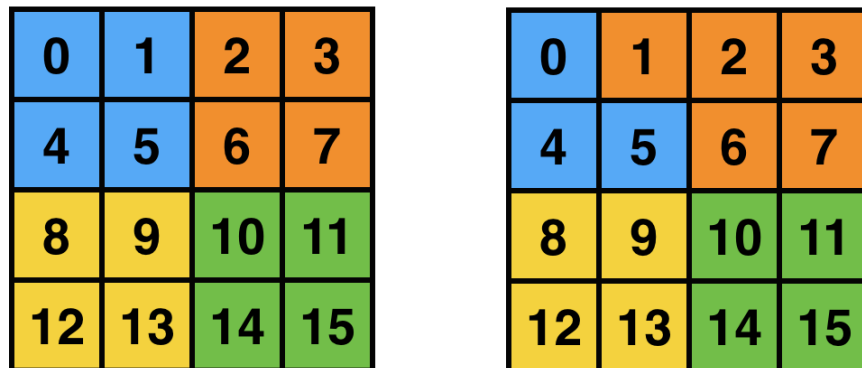
One important concern is that of load imbalance. New generation parallel applications are dynamically varying, meaning that processors’ load is shifting during execution. In a dynamic simulation application such as rocket simulation,



4: The problem domain is over-decomposed to more VPs than PEs.

burning solid fuel, sub-scaling for a certain part of the mesh, crack propagation, particle flows all contribute to load imbalance. A centralized load balancing strategy built into an application is impractical since each individual module is developed mostly independently by various developers. In addition, embedding a load balancing strategy in the code complicates it greatly, and programming effort increases significantly. The runtime system is uniquely positioned to deal with load imbalance. Figure 5 shows the runtime system migrating a VP after detecting load imbalance. This domain may correspond to a weather forecast model where there is a storm cell in the top-left quadrant, which requires more computation to simulate. AMPI will then migrate VP 1 to balance the division of work across processors and improve performance. Note that incorporating this sort of load balancing inside the application code may take a lot of effort and complicate the code.

Migration of VP 1



5: AMPI can migrate VPs across processes for load balancing.

There are many different load balancing strategies built into Charm++ that can be selected by an AMPI application developer. Among those, some may fit better for a particular application depending on its characteristics. Moreover, one can write a new load balancer, best suited for an application, by the simple API provided inside Charm++ infrastructure. Our approach is based on actual measurement of load information at runtime, and on migrating computations from heavily loaded to lightly loaded processors.

For this approach to be effective, we need the computation to be split into pieces many more in number than available processors. This allows us to flexibly map and re-map these computational pieces to available processors. This approach is usually called “multi-domain decomposition”.

Charm++, which we use as a runtime system layer for the work described here, simplifies our approach. It embeds an

elaborate performance tracing mechanism, a suite of plug-in load balancing strategies, infrastructure for defining and migrating computational load, and is interoperable with other programming paradigms.

3.1.2 Charm++

Charm++ is an object-oriented parallel programming library for C. It differs from traditional message passing programming libraries (such as MPI) in that Charm++ is “message-driven”. Message-driven parallel programs do not block the processor waiting for a message to be received. Instead, each message carries with itself a computation that the processor performs on arrival of that message. The underlying runtime system of Charm++ is called Converse, which implements a “scheduler” that chooses which message to schedule next (message-scheduling in Charm++ involves locating the object for which the message is intended, and executing the computation specified in the incoming message on that object). A parallel object in Charm++ is a C object on which a certain computations can be asked to be performed from remote processors.

Charm++ programs exhibit latency tolerance since the scheduler always picks up the next available message rather than waiting for a particular message to arrive. They also tend to be modular, because of their object-based nature. Most importantly, Charm++ programs can be *dynamically load balanced*, because the messages are directed at objects and not at processors; thus allowing the runtime system to migrate the objects from heavily loaded processors to lightly loaded processors.

Since many CSE applications are originally written using MPI, one would have to rewrite existing code if they were to be converted to Charm++ to take advantage of dynamic load balancing and other Charm++ features. This is indeed impractical. However, Converse - the runtime system of Charm++ - supports interoperability between different parallel programming paradigms such as parallel objects and threads. Using this feature, we developed AMPI, which is described in more detail in the next section.

3.1.3 AMPI

AMPI utilizes the dynamic load balancing and other capabilities of Charm++ by associating a “user-level” thread with each Charm++ migratable object. User’s code runs inside this thread, so that it can issue blocking receive calls similar to MPI, and still present the underlying scheduler an opportunity to schedule other computations on the same processor. The runtime system keeps track of the computational loads of each thread as well as the communication graph between AMPI threads, and can migrate these threads in order to balance the overall load while simultaneously minimizing communication overhead.

MPI Standards Compliance

Currently AMPI supports the MPI-2.2 standard, with preliminary support for most MPI-3.1 features and a collection of extensions explained in detail in this manual. One-sided communication calls in MPI-2 and MPI-3 are implemented, but they do not yet take advantage of RMA features. Non-blocking collectives have been defined in AMPI since before MPI-3.0’s adoption of them. ROMIO (<http://www-unix.mcs.anl.gov/romio/>) has been integrated into AMPI to support parallel I/O features.

3.2 Building and Running AMPI Programs

3.2.1 Installing AMPI

AMPI is included in the source distribution of Charm++. To get the latest sources from PPL, visit: <https://charm.cs.illinois.edu/software>

and follow the download links. Then build Charm++ and AMPI from source.

The build script for Charm++ is called `build`. The syntax for this script is:

```
$ ./build <target> <version> <opts>
```

Users who are interested only in AMPI and not any other component of Charm++ should specify `<target>` to be `AMPI-only`. This will build Charm++ and other libraries needed by AMPI in a mode configured and tuned exclusively for AMPI. To fully build Charm++ underneath AMPI for use with either paradigm, or for interoperation between the two, specify `<target>` to be `AMPI`.

`<opts>` are command line options passed to the `charmcc` compile script. Common compile time options such as `-g`, `-O`, `-Ipath`, `-Lpath`, `-llib` are accepted.

To build a debugging version of AMPI, use the option: `-g`. To build a production version of AMPI, use the option: `--with-production`.

`<version>` depends on the machine, operating system, and the underlying communication library one wants to use for running AMPI programs. See the `charm/README` file for details on picking the proper version. Here is an example of how to build a debug version of AMPI in a linux and ethernet environment:

```
$ ./build AMPI netlrts-linux-x86_64 -g
```

And the following is an example of how to build a production version of AMPI on a Cray XC system, with MPI-level error checking in AMPI turned off:

```
$ ./build AMPI-only gni-crayxc --with-production --disable-mpi-error-checking
```

AMPI can also be built with support for multithreaded parallelism on any communication layer by adding “`smp`” as an option after the build target. For example, on an Infiniband Linux cluster:

```
$ ./build AMPI-only verbs-linux-x86_64 smp --with-production
```

AMPI ranks are implemented as user-level threads with a stack size default of 1MB. If the default is not correct for your program, you can specify a different default stack size (in bytes) at build time. The following build command illustrates this for an Intel Omni-Path system:

```
$ ./build AMPI-only ofi-linux-x86_64 --with-production -DTCHARM_STACKSIZE_
↪DEFAULT=16777216
```

The same can be done for AMPI’s RDMA messaging threshold using `AMPI_RDMA_THRESHOLD_DEFAULT` and, for messages sent within the same address space (ranks on the same worker thread or ranks on different worker threads in the same process in SMP builds), using `AMPI_SMP_RDMA_THRESHOLD_DEFAULT`. Contiguous messages with sizes larger than the threshold are sent via RDMA on communication layers that support this capability. You can also set the environment variables `AMPI_RDMA_THRESHOLD` and `AMPI_SMP_RDMA_THRESHOLD` before running a job to override the default specified at build time.

3.2.2 Building AMPI Programs

AMPI provides compiler wrappers such as `ampicc`, `ampif90`, and `ampicxx` in the `bin` subdirectory of Charm++ installations. You can use them to build your AMPI program using the same syntax as other compilers like `gcc`. They are intended as drop-in replacements for `mpicc` wrappers provided by most conventional MPI implementations. These scripts automatically handle the details of building and linking against AMPI and the Charm++ runtime system. This includes launching the compiler selected during the Charm++ build process, passing any toolchain parameters important for proper function on the selected build target, supplying the include and link paths for the runtime system, and linking with Charm++ components important for AMPI, including Isomalloc heap interception and commonly used load balancers.

3: Full list of AMPI toolchain wrappers.

Command Name	Purpose
mpicc	C
mpiCC	C++
mpicxx	C++
mpic++	C++
mpif77	Fortran 77
mpif90	Fortran 90
mpifort	Fortran 90
mpirun	Program Launch
mpiexec	Program Launch

All command line flags that you would use for other compilers can be used with the AMPI compilers the same way. For example:

```
$ mpicc -c pgm.c -O3
$ mpif90 -c pgm.f90 -O0 -g
$ mpicc -o pgm pgm.o -lm -O3
```

For consistency with other MPI implementations, these wrappers are also provided using their standard names with the suffix `.mpi`:

```
$ mpicc.mpi -c pgm.c -O3
$ mpif90.mpi -c pgm.f90 -O0 -g
$ mpicc.mpi -o pgm pgm.o -lm -O3
```

Additionally, the `bin/mpi` subdirectory of Charm++ installations contains the wrappers with their exact standard names, allowing them to be given precedence as shell commands in a `module`-like fashion by adding this directory to the `$PATH` environment variable:

```
$ export PATH=/home/user/charm/netlrts-linux-x86_64/bin/mpi:$PATH $ mpicc -c pgm.c -O3 $ mpif90
-c pgm.f90 -O0 -g $ mpicc -o pgm pgm.o -lm -O3
```

These wrappers also allow the user to configure AMPI and Charm++-specific functionality. For example, to automatically select a Charm++ load balancer at program launch without passing the `+balancer` runtime parameter, specify a strategy at link time with `-balancer <LB>`:

```
$ mpicc pgm.c -o pgm -O3 -balancer GreedyRefineLB
```

Internally, the toolchain wrappers call the Charm runtime's general toolchain script, `charm.c`. By default, they will specify `-memory isomalloc` and `-module CommonLBs`. Advanced users can disable Isomalloc heap interception by passing `-memory default`. For diagnostic purposes, the `-verbose` option will print all parameters passed to each stage of the toolchain. Refer to the Charm++ manual for information about the full set of parameters supported by `charm.c`.

3.2.3 Running AMPI Programs

AMPI offers two options to execute an AMPI program, `charmrun` and `mpirun`.

Running with `charmrun`

The Charm++ distribution contains a script called `charmrun` that makes the job of running AMPI programs portable and easier across all parallel machines supported by Charm++. `charmrun` is copied to a directory where an AMPI

program is built using `mpicc`. It takes a command line parameter specifying number of processors, and the name of the program followed by AMPI options (such as number of ranks to create, and the stack size of every user-level thread) and the program arguments. A typical invocation of an AMPI program `pgm` with `charmrun` is:

```
$ ./charmrun +p16 ./pgm +vp64
```

Here, the AMPI program `pgm` is run on 16 physical processors with 64 total virtual ranks (which will be mapped 4 per processor initially).

To run with load balancing, specify a load balancing strategy.

You can also specify the size of user-level thread's stack using the `+tcharm_stacksize` option, which can be used to decrease the size of the stack that must be migrated, as in the following example:

```
$ ./charmrun +p16 ./pgm +vp128 +tcharm_stacksize 32K +balancer RefineLB
```

Running with ampirun

For compliance with the MPI standard and simpler execution, AMPI ships with the `ampirun` script that is similar to `mpirun` provided by other MPI runtimes. As with `charmrun`, `ampirun` is copied automatically to the program directory when compiling an application with `mpicc`. Users with prior MPI experience may find `ampirun` the simplest way to run AMPI programs.

The basic usage of `ampirun` is as follows:

```
$ ./ampirun -np 16 --host h1,h2,h3,h4 ./pgm
```

This command will create 16 (non-virtualized) ranks and distribute them on the hosts `h1-h4`.

When using the `-vr` option, AMPI will create the number of ranks specified by the `-np` parameter as virtual ranks, and will create only one process per host:

```
$ ./ampirun -np 16 --host h1,h2,h3,h4 -vr ./pgm
```

Other options (such as the load balancing strategy), can be specified in the same way as for `charmrun`:

```
$ ./ampirun -np 16 ./pgm +balancer RefineLB
```

Other options

Note that for AMPI programs compiled with `gfortran`, users may need to set the following environment variable to see program output on `stdout`:

```
$ export GFORTRAN_UNBUFFERED_ALL=1
```

3.3 Using Existing MPI Codes with AMPI

Due to the nature of AMPI's virtualized ranks, some changes to existing MPI codes may be necessary for them to function correctly with AMPI.

3.3.1 Entry Point

To convert an existing program to use AMPI, the main function or program may need to be renamed. The changes should be made as follows:

Fortran

You must declare the main program as a subroutine called “MPI_MAIN”. Do not declare the main subroutine as a *program* because it will never be called by the AMPI runtime.

```
program pgm -> subroutine MPI_Main
...
end program -> end subroutine
```

C or C++

The main function can be left as is, if `mpi.h` is included before the main function. This header file has a preprocessor macro that renames `main`, and the renamed version is called by the AMPI runtime for each rank.

3.3.2 Command Line Argument Parsing

Fortran

For parsing Fortran command line arguments, AMPI Fortran programs should use our extension APIs, which are similar to Fortran 2003’s standard APIs. For example:

```
integer :: i, argc, ierr
integer, parameter :: arg_len = 128
character(len=arg_len), dimension(:), allocatable :: raw_arguments

call AMPI_Command_argument_count(argc)
allocate(raw_arguments(argc))
do i = 1, size(raw_arguments)
    call AMPI_Get_command_argument(i, raw_arguments(i), arg_len, ierr)
end do
```

C or C++

Existing code for parsing `argc` and `argv` should be sufficient, provided that it takes place *after* `MPI_Init`.

3.3.3 Global Variable Privatization

In AMPI, ranks are implemented as user-level threads that coexist within OS processes or OS threads, depending on how the Charm++ runtime was built. Traditional MPI programs assume that each rank has an entire OS process to itself, and that only one thread of control exists within its address space. This allows them to safely use global and static variables in their code. However, global and static variables are problematic for multi-threaded environments such as AMPI or OpenMP. This is because there is a single instance of those variables, so they will be shared among different ranks in the single address space, and this could lead to the program producing an incorrect result or crashing.

The following code is an example of this problem. Each rank queries its numeric ID, stores it in a global variable, waits on a global barrier, and then prints the value that was stored. If this code is run with multiple ranks virtualized

inside one OS process, each rank will store its ID in the same single location in memory. The result is that all ranks will print the ID of whichever one was the last to successfully update that location. For this code to be semantically valid with AMPI, each rank needs its own separate instance of the variable. This is where the need arises for some special handling of these unsafe variables in existing MPI applications, which we call *privatization*.

```
int rank_global;

void print_ranks(void)
{
    MPI_Comm_rank(MPI_COMM_WORLD, &rank_global);

    MPI_Barrier(MPI_COMM_WORLD);

    printf("rank: %d\n", rank_global);
}
```

The basic transformation needed to port MPI programs to AMPI is privatization of global and static variables. Module variables, “saved” subroutine local variables, and common blocks in Fortran90 also belong to this category. Certain API calls use global variables internally, such as `strtok` in the C standard library, and as a result they are also unsafe. If such a program is executed without privatization on AMPI, all the AMPI ranks that reside in the same process will access the same copy of such variables, which is clearly not the desired semantics. Note that global variables that are constant or are only written to once during initialization with the same value across all ranks are already thread-safe.

To ensure AMPI programs execute correctly, it is necessary to make such variables “private” to individual ranks. We provide several options to achieve this with varying degrees of portability and required developer effort.

Warning: If you are writing a new MPI application from scratch and would like to support AMPI as a first-class target, it is highly recommended to follow certain guidelines for writing your code to avoid the global variable problem entirely, eliminating the need for time-consuming refactoring or platform-specific privatization methods later on. See the Manual Code Editing section below for an example of how to structure your code in order to accomplish this.

Manual Code Editing

With regard to performance and portability, the ideal approach to resolve the global variable problem is to refactor your code to avoid use of globals entirely. However, this comes with the obvious caveat that it requires developer time to implement and can involve invasive changes across the entire codebase, similar to converting a shared library to be reentrant in order to allow multiple instantiations from the same OS process. If these costs are a significant barrier to entry, it can be helpful to instead explore one of the simpler transformations or fully automated methods described below.

We have employed a strategy of argument passing to do this privatization transformation. That is, the global variables are bunched together in a single user-defined type, which is allocated by each thread dynamically or on the stack. Then a pointer to this type is passed from subroutine to subroutine as an argument. Since the subroutine arguments are passed on the stack, which is not shared across all threads, each subroutine when executing within a thread operates on a private copy of the global variables.

This scheme is demonstrated in the following examples. The original Fortran90 code contains a module `shareddata`. This module is used in the `MPI_MAIN` subroutine and a subroutine `subA`. Note that `PROGRAM PGM` was renamed to `SUBROUTINE MPI_MAIN` and `END PROGRAM` was renamed to `END SUBROUTINE`.

```
!FORTRAN EXAMPLE
MODULE shareddata
    INTEGER :: myrank
```

(continues on next page)

(continued from previous page)

```

    DOUBLE PRECISION :: xyz(100)
END MODULE

SUBROUTINE MPI_MAIN                                ! Previously PROGRAM PGM
    USE shareddata
    include 'mpif.h'
    INTEGER :: i, ierr
    CALL MPI_Init(ierr)
    CALL MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
    DO i = 1, 100
        xyz(i) = i + myrank
    END DO
    CALL subA
    CALL MPI_Finalize(ierr)
END SUBROUTINE                                ! Previously END PROGRAM

SUBROUTINE subA
    USE shareddata
    INTEGER :: i
    DO i = 1, 100
        xyz(i) = xyz(i) + 1.0
    END DO
END SUBROUTINE

```

```

//C Example
#include <mpi.h>

int myrank;
double xyz[100];

void subA();
int main(int argc, char** argv){
    int i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for(i=0;i<100;i++)
        xyz[i] = i + myrank;
    subA();
    MPI_Finalize();
}

void subA(){
    int i;
    for(i=0;i<100;i++)
        xyz[i] = xyz[i] + 1.0;
}

```

AMPI executes the main subroutine inside a user-level thread as a subroutine.

Now we transform this program using the argument passing strategy. We first group the shared data into a user-defined type.

```

!FORTRAN EXAMPLE
MODULE shareddata
    TYPE chunk ! modified
        INTEGER :: myrank

```

(continues on next page)

(continued from previous page)

```

    DOUBLE PRECISION :: xyz(100)
    END TYPE ! modified
END MODULE

```

```

//C Example
struct shareddata{
    int myrank;
    double xyz[100];
};

```

Now we modify the main subroutine to dynamically allocate this data and change the references to them. Subroutine subA is then modified to take this data as argument.

```

!FORTRAN EXAMPLE
SUBROUTINE MPI_Main
    USE shareddata
    USE AMPI
    INTEGER :: i, ierr
    TYPE(chunk), pointer :: c ! modified
    CALL MPI_Init(ierr)
    ALLOCATE(c) ! modified
    CALL MPI_Comm_rank(MPI_COMM_WORLD, c%myrank, ierr)
    DO i = 1, 100
        c%xyz(i) = i + c%myrank ! modified
    END DO
    CALL subA(c)
    CALL MPI_Finalize(ierr)
END SUBROUTINE

SUBROUTINE subA(c)
    USE shareddata
    TYPE(chunk) :: c ! modified
    INTEGER :: i
    DO i = 1, 100
        c%xyz(i) = c%xyz(i) + 1.0 ! modified
    END DO
END SUBROUTINE

```

```

//C Example
void MPI_Main{
    int i,ierr;
    struct shareddata *c;
    ierr = MPI_Init();
    c = (struct shareddata*)malloc(sizeof(struct shareddata));
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, c->myrank);
    for(i=0;i<100;i++)
        c->xyz[i] = i + c->myrank;
    subA(c);
    ierr = MPI_Finalize();
}

void subA(struct shareddata *c){
    int i;
    for(i=0;i<100;i++)
        c->xyz[i] = c->xyz[i] + 1.0;
}

```

With these changes, the above program can be made thread-safe. Note that it is not really necessary to dynamically allocate `chunk`. One could have declared it as a local variable in subroutine `MPI_Main`. (Or for a small example such as this, one could have just removed the `shareddata` module, and instead declared both variables `xyz` and `myrank` as local variables). This is indeed a good idea if shared data are small in size. For large shared data, it would be better to do heap allocation because in AMPI, the stack sizes are fixed at the beginning (and can be specified from the command line) and stacks do not grow dynamically.

Automatic Thread-Local Storage Swapping

Thread Local Store (TLS) was originally employed in kernel threads to localize variables to threads and provide thread safety. It can be used by annotating global/static variable declarations in C with `thread_local`, in C with `__thread` or C11 with `thread_local` or `_Thread_local`, and in Fortran with OpenMP's `threadprivate` attribute. OpenMP is required for using `tlsglobals` in Fortran code since Fortran has no other method of using TLS. The `__thread` keyword is not an official extension of the C language, though compiler writers are encouraged to implement this feature.

It handles both global and static variables and has no context-switching overhead. AMPI provides runtime support for privatizing thread-local variables to user-level threads by changing the TLS segment register when context switching between user-level threads. The runtime overhead is that of changing a single pointer per user-level thread context switch. Currently, Charm++ supports it for x86/x86_64 platforms when using GNU compilers.

```
// C/C++ example:
int myrank;
double xyz[100];
```

```
! Fortran example:
integer :: myrank
real*8, dimension(100) :: xyz
```

For the example above, the following changes to the code handle the global variables:

```
// C++ example:
thread_local int myrank;
thread_local double xyz[100];

// C example:
__thread int myrank;
__thread double xyz[100];
```

```
! Fortran example:
integer :: myrank
real*8, dimension(100) :: xyz
!$omp threadprivate(myrank)
!$omp threadprivate(xyz)
```

The runtime system also should know that TLS-Globals is used at both compile and link time:

```
$ ampicxx -o example example.C -tlsglobals
```

Automatic Process-in-Process Runtime Linking Privatization

Process-in-Process (PiP) [PiP2018] Globals allows fully automatic privatization of global variables on GNU/Linux systems without modification of user code. All languages (C, C++, Fortran, etc.) are supported. This method currently lacks support for checkpointing and migration, which are necessary for load balancing and fault tolerance.

Additionally, overdecomposition is limited to approximately 12 virtual ranks per logical node, though this can be resolved by building a patched version of glibc.

This method works by combining a specific method of building binaries with a GNU extension to the dynamic linker. First, AMPI's toolchain wrapper compiles your user program as a Position Independent Executable (PIE) and links it against a special shim of function pointers instead of the normal AMPI runtime. It then builds a small loader utility that links directly against AMPI. For each rank, this loader calls the glibc-specific function `dlopen` on the PIE binary with a unique namespace index. The loader uses `dlsym` to populate the PIE binary's function pointers and then it calls the entry point. This `dlopen` and `dlsym` process repeats for each rank. As soon as execution jumps into the PIE binary, any global variables referenced within will appear privatized. This is because PIE binaries locate the global data segment immediately after the code segment so that PIE global variables are accessed relative to the instruction pointer, and because `dlopen` creates a separate copy of these segments in memory for each unique namespace index.

Optionally, the first step in using PiP-Globals is to build PiP-glibc to overcome the limitation on rank count per process. Use the instructions at <https://github.com/RIKEN-SysSoft/PiP/blob/pip-1/INSTALL.md> to download an installable PiP package or build PiP-glibc from source by following the Patched GLIBC section. AMPI may be able to automatically detect PiP's location if installed as a package, but otherwise set and export the environment variable `PIP_GLIBC_INSTALL_DIR` to the value of `<GLIBC_INSTALL_DIR>` as used in the above instructions. For example:

```
$ export PIP_GLIBC_INSTALL_DIR=~/.pip
```

To use PiP-Globals in your AMPI program (with or without PiP-glibc), compile and link with the `-pipglobals` parameter:

```
$ ampicxx -o example.o -c example.cpp -pipglobals
$ ampicxx -o example example.o -pipglobals
```

No further effort is needed. Global variables in `example.cpp` will be automatically privatized when the program is run. Any libraries and shared objects compiled as PIE will also be privatized. However, if these objects call MPI functions, it will be necessary to build them with the AMPI toolchain wrappers, `-pipglobals`, and potentially also the `-standalone` parameter in the case of shared objects. It is recommended to do this in any case so that AMPI can ensure everything is built as PIE.

Potential future support for checkpointing and migration will require modification of the `ld-linux.so` runtime loader to intercept mmap allocations of the previously mentioned segments and redirect them through `Isomalloc`. The present lack of support for these features mean PiP-Globals is best suited for testing AMPI during exploratory phases of development, and for production jobs not requiring load balancing or fault tolerance.

Automatic Filesystem-Based Runtime Linking Privatization

Filesystem Globals (FS-Globals) was discovered during the development of PiP-Globals and the two are highly similar. Like PiP-Globals, it requires no modification of user code and works with any language. It also currently lacks support for checkpointing and migration, preventing use of load balancing and fault tolerance. Unlike PiP-Globals, it is portable beyond GNU/Linux and has no limits to overdecomposition beyond available disk space.

FS-Globals works in the same way as PiP-Globals except that instead of specifying namespaces using `dlopen`, which is a GNU/Linux-specific feature, this method creates copies of the user's PIE binary on the filesystem for each rank and calls the POSIX-standard `dlopen`.

To use FS-Globals, compile and link with the `-fsglobals` parameter:

```
$ ampicxx -o example.o -c example.cpp -fsglobals
$ ampicxx -o example example.o -fsglobals
```

No additional steps are required. Global variables in `example.cpp` will be automatically privatized when the program is run. Variables in statically linked libraries will also be privatized if compiled as PIE. It is recommended to achieve this by building with the AMPI toolchain wrappers and `-fsglobals`, and this is necessary if the libraries call MPI functions. Shared objects are currently not supported by FS-Globals due to the extra overhead of iterating through all dependencies and copying each one per rank while avoiding system components, plus the complexity of ensuring each rank's program binary sees the proper set of objects.

This method's use of the filesystem is a drawback in that it is slow during startup and can be considered wasteful. Additionally, support for load balancing and fault tolerance would require further development in the future, using the same infrastructure as what PiP-Globals would require. For these reasons FS-Globals is best suited for the R&D phase of AMPI program development and for small jobs, and it may be less suitable for large production environments.

Automatic Global Offset Table Swapping

Thanks to the ELF Object Format, we have successfully automated the procedure of switching the set of user global variables when switching thread contexts. Executable and Linkable Format (ELF) is a common standard file format for Object Files in Unix-like operating systems. ELF maintains a Global Offset Table (GOT) for globals so it is possible to switch GOT contents at thread context-switch by the runtime system.

The only thing that the user needs to do is pass the flag `-swapglobals` at both compile and link time (e.g. “`amputc -o prog prog.c -swapglobals`”). This method does not require any changes to the source code and works with any language (C, C++, Fortran, etc). However, it does not handle static variables, has a context switching overhead that grows with the number of global variables, and is incompatible with SMP builds of AMPI, where multiple virtual ranks can execute simultaneously on different scheduler threads within an OS process.

Currently, this feature only works on x86 and x86_64 platforms that fully support ELF, and it requires `ld` version 2.23 or older, or else a patched version of `ld` 2.24+ that we provide here: <https://charm.cs.illinois.edu/gerrit/gitweb?p=libbfd-patches.git;a=tree;f=swapglobals>

For these reasons, and because more robust privatization methods are available, `swapglobals` is considered deprecated.

Source-to-Source Transformation

One final approach is to use a tool to transform your program's source code, implementing the changes described in one of the sections above in an automated fashion.

We have multiple tools for automating these transformations for different languages. Currently, there is a tool called *Photran* (<http://www.eclipse.org/photran>) for refactoring Fortran codes that can do this transformation. It is Eclipse-based and works by constructing Abstract Syntax Trees (ASTs) of the program. We also have a tool built with *LLVM/LibTooling* that applies the TLS-Globals transformation to C/C++ codes, available upon request.

Summary

Table 4 shows portability of different schemes.

4: Portability of current implementations of three privatization schemes. “Yes” means we have implemented this technique. “Maybe” indicates there are no theoretical problems, but no implementation exists. “No” indicates the technique is impossible on this platform.

Privatization Scheme	Linux	Mac OS	BG/Q	Windows	x86	x86_64	PPC	ARM7
Manual Code Editing	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
TLS-Globals	Yes	Yes	No	Maybe	Yes	Yes	Maybe	Maybe
PiP-Globals	Yes	No	No	No	Yes	Yes	Yes	Yes
FS-Globals	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
GOT-Globals	Yes	No	No	No	Yes	Yes	Yes	Yes

3.4 Extensions

The following are AMPI extensions to the MPI standard, which will be explained in detail in this manual. All AMPI extensions to the MPI standard are prefixed with `AMPI_` rather than `MPI_`. All extensions are available in C, C++, and Fortran, with the exception of `AMPI_Command_argument_count` and `AMPI_Get_command_argument` which are only available in Fortran.

<code>AMPI_Migrate</code>	<code>AMPI_Register_pup</code>	<code>AMPI_Get_pup_data</code>
<code>AMPI_Migrate_to_pe</code>	<code>AMPI_Set_migratable</code>	
<code>AMPI_Load_set_value</code>	<code>AMPI_Load_start_measure</code>	<code>AMPI_Load_stop_measure</code>
<code>AMPI_Iget</code>	<code>AMPI_Iget_wait</code>	<code>AMPI_Iget_data</code>
<code>AMPI_Iget_free</code>	<code>AMPI_Type_is_contiguous</code>	
<code>AMPI_Yield</code>	<code>AMPI_Suspend</code>	<code>AMPI_Resume</code>
<code>AMPI_Alltoall_medium</code>	<code>AMPI_Alltoall_long</code>	
<code>AMPI_Register_just_migrated</code>	<code>AMPI_Register_about_to_migrate</code>	
<code>AMPI_Command_argument_count</code>	<code>AMPI_Get_command_argument</code>	

3.4.1 Serialization

Some of AMPI’s primary benefits are made possible by the ability to pack and unpack the entire state of a program and transmit it over the network or write a snapshot of it to the filesystem.

In the vast majority of cases, this serialization is fully automated using a custom memory allocator, `Isomalloc`, which returns virtual memory addresses that are globally unique across an entire job. This means that every worker thread in the system reserves slices of virtual memory for all user-level threads, allowing transparent migration of stacks and pointers into memory. (`Isomalloc` requires 64-bit virtual memory addresses and support from the operating system for mapping memory to arbitrary virtual addresses.) Applications built with AMPI’s toolchain wrappers are automatically linked with `Isomalloc` as the active `malloc` implementation if the target platform supports the feature.

For systems that do not support `Isomalloc` and for users that wish to have more fine-grain control over which application data structures will be copied at migration time, we have added a few calls to AMPI. These include the ability to register thread-specific data with the run-time system, and the means to pack and unpack all of the thread’s data. This mode of operation requires passing `-memory_default` at link time to disable `Isomalloc`’s heap interception.

Warning: Most users may skip this section unless you have specific needs.

AMPI packs up any data internal to the runtime in use by the rank, including the thread’s stack. This means that the local variables declared in subroutines in a rank, which are created on the stack, are automatically packed by the runtime system. However, without `Isomalloc`, the runtime has no way of knowing what other data are in use by the

rank. Thus upon starting execution, a rank needs to notify the system about the data that it is going to use (apart from local variables). Even with the data registration, AMPI cannot determine what size the data is, or whether the registered data contains pointers to other places in memory. For this purpose, a packing subroutine also needs to be provided to the AMPI runtime system along with registered data. The call provided by AMPI for doing this is `AMPI_Register_pup`. This function takes three arguments: a data item to be transported along with the rank, the pack subroutine, and a pointer to an integer which denotes the registration identifier. In C/C++ programs, it may be necessary to use this integer value after migration completes and control returns to the rank with the function `AMPI_Get_pup_data`.

Once the AMPI runtime system decides which ranks to send to which processors, it calls the specified pack subroutine for that rank, with the rank-specific data that was registered with the system using `AMPI_Register_pup`. If an AMPI application uses Isomalloc, then the system will define the Pack/Unpack routines for the user. This section explains how a subroutine should be written for performing explicit pack/unpack.

There are three steps for transporting the rank's data to another processor. First, the system calls a subroutine to get the size of the buffer required to pack the rank's data. This is called the "sizing" step. In the next step, which is called immediately afterward on the source processor, the system allocates the required buffer and calls the subroutine to pack the rank's data into that buffer. This is called the "packing" step. This packed data is then sent as a message to the destination processor, where first a rank is created (along with the thread) and a subroutine is called to unpack the rank's data from the buffer. This is called the "unpacking" step.

Though the above description mentions three subroutines called by the AMPI runtime system, it is possible to actually write a single subroutine that will perform all the three tasks. This is achieved using something we call a "pupper". A pupper is an external subroutine that is passed to the rank's pack-unpack-sizing subroutine, and this subroutine, when called in different phases performs different tasks. An example will make this clear:

Suppose the user data, chunk, is defined as a derived type in Fortran90:

```
!FORTRAN EXAMPLE
MODULE chunkmod
  INTEGER, parameter :: nx=4, ny=4, tchunks=16
  TYPE, PUBLIC :: chunk
    REAL(KIND=8) t(22,22)
    INTEGER xidx, yidx
    REAL(KIND=8), dimension(400) :: bxm, bxp, bym, byp
  END TYPE chunk
END MODULE
```

```
//C Example
struct chunk{
  double t;
  int xidx, yidx;
  double bxm,bxp,bym,byp;
};
```

Then the pack-unpack subroutine `chunkpup` for this chunk module is written as:

```
!FORTRAN EXAMPLE
SUBROUTINE chunkpup(p, c)
  USE pupmod
  USE chunkmod
  IMPLICIT NONE
  INTEGER :: p
  TYPE(chunk) :: c

  call pup(p, c%t)
  call pup(p, c%xidx)
```

(continues on next page)

(continued from previous page)

```

call pup(p, c%yidx)
call pup(p, c%bxm)
call pup(p, c%bxp)
call pup(p, c%bym)
call pup(p, c%byp)
end subroutine

```

```

//C Example
void chunkpup(pup_er p, struct chunk c){
    pup_double(p,c.t);
    pup_int(p,c.xidx);
    pup_int(p,c.yidx);
    pup_double(p,c.bxm);
    pup_double(p,c.bxp);
    pup_double(p,c.bym);
    pup_double(p,c.byp);
}

```

There are several things to note in this example. First, the same subroutine `pup` (declared in module `pupmod`) is called to size/pack/unpack any type of data. This is possible because of procedure overloading possible in Fortran90. Second is the integer argument `p`. It is this argument that specifies whether this invocation of subroutine `chunkpup` is sizing, packing or unpacking. Third, the integer parameters declared in the type `chunk` need not be packed or unpacked since they are guaranteed to be constants and thus available on any processor.

A few other functions are provided in module `pupmod`. These functions provide more control over the packing/unpacking process. Suppose one modifies the `chunk` type to include allocatable data or pointers that are allocated dynamically at runtime. In this case, when `chunk` is packed, these allocated data structures should be deallocated after copying them to buffers, and when `chunk` is unpacked, these data structures should be allocated before copying them from the buffers. For this purpose, one needs to know whether the invocation of `chunkpup` is a packing one or unpacking one. For this purpose, the `pupmod` module provides functions `fpup_isdeleting`(`fpup_isunpacking`). These functions return logical value `.TRUE.` if the invocation is for packing (unpacking), and `.FALSE.` otherwise. The following example demonstrates this:

Suppose the type `dchunk` is declared as:

```

!FORTRAN EXAMPLE
MODULE dchunkmod
  TYPE, PUBLIC :: dchunk
    INTEGER :: asize
    REAL(KIND=8), pointer :: xarr(:), yarr(:)
  END TYPE dchunk
END MODULE

```

```

//C Example
struct dchunk{
    int asize;
    double* xarr, *yarr;
};

```

Then the pack-unpack subroutine is written as:

```

!FORTRAN EXAMPLE
SUBROUTINE dchunkpup(p, c)
  USE pupmod
  USE dchunkmod

```

(continues on next page)

(continued from previous page)

```

IMPLICIT NONE
INTEGER :: p
TYPE(dchunk) :: c

pup(p, c%asize)

IF (fpup_isunpacking(p)) THEN           !! if invocation is for unpacking
    allocate(c%xarr(c%asize))
    ALLOCATE(c%yarr(c%asize))
ENDIF

pup(p, c%xarr)
pup(p, c%yarr)

IF (fpup_isdeleting(p)) THEN           !! if invocation is for packing
    DEALLOCATE(c%xarr)
    DEALLOCATE(c%yarr)
ENDIF

END SUBROUTINE

```

```

//C Example
void dchunkpup(pup_er p, struct dchunk c){
    pup_int(p,c.asize);
    if(pup_isUnpacking(p)){
        c.xarr = (double *)malloc(sizeof(double)*c.asize);
        c.yarr = (double *)malloc(sizeof(double)*c.asize);
    }
    pup_doubles(p,c.xarr,c.asize);
    pup_doubles(p,c.yarr,c.asize);
    if(pup_isPacking(p)){
        free(c.xarr);
        free(c.yarr);
    }
}

```

One more function `fpup_issizing` is also available in module `pupmod` that returns `.TRUE.` when the invocation is a sizing one. In practice one almost never needs to use it.

Charm++ also provides higher-level PUP routines for C++ STL data structures and Fortran90 data types. The STL PUP routines will deduce the size of the structure automatically, so that the size of the data does not have to be passed in to the PUP routine. This facilitates writing PUP routines for large pre-existing codebases. To use it, simply include `pup_stl.h` in the user code. For modern Fortran with pointers and allocatable data types, AMPI provides a similarly automated PUP interface called `apup`. User code can include `pupmod` and then call `apup()` on any array (pointer or allocatable, multi-dimensional) of built-in types (character, short, int, long, real, double, complex, double complex, logical) and the runtime will deduce the size and shape of the array, including unassociated and NULL pointers. Here is the `dchunk` example from earlier, written to use the `apup` interface:

```

!FORTRAN EXAMPLE
SUBROUTINE dchunkpup(p, c)
    USE pupmod
    USE dchunkmod
    IMPLICIT NONE
    INTEGER :: p
    TYPE(dchunk) :: c

```

(continues on next page)

(continued from previous page)

```

!! no need for asize
!! no isunpacking allocation necessary

apup(p, c%xarr)
apup(p, c%yarr)

!! no isdeleting deallocation necessary

END SUBROUTINE

```

Calling `MPI_` routines or accessing global variables that have been privatized by use of `tlsglobals` or `swapglobals` from inside a user PUP routine is currently not allowed in AMPI. Users can store MPI-related information like communicator rank and size in data structures to be packed and unpacked before they are needed inside a PUP routine.

3.4.2 Load Balancing and Migration

AMPI provides support for migrating MPI ranks between nodes of a system. If the AMPI runtime system is prompted to examine the distribution of work throughout the job and decides that load imbalance exists within the application, it will invoke one of its internal load balancing strategies, which determines the new mapping of AMPI ranks so as to balance the load. Then the AMPI runtime serializes the rank's state as described above and moves it to its new home processor.

AMPI provides a subroutine `AMPI_Migrate(MPI_Info hints)`; for this purpose. Each rank periodically calls `AMPI_Migrate`. Typical CSE applications are iterative and perform multiple time-steps. One should call `AMPI_Migrate` in each rank at the end of some fixed number of timesteps. The frequency of `AMPI_Migrate` should be determined by a tradeoff between conflicting factors such as the load balancing overhead, and performance degradation caused by load imbalance. In some other applications, where application suspects that load imbalance may have occurred, as in the case of adaptive mesh refinement; it would be more effective if it performs a couple of timesteps before telling the system to re-map ranks. This will give the AMPI runtime system some time to collect the new load and communication statistics upon which it bases its migration decisions. Note that `AMPI_Migrate` does NOT tell the system to migrate the rank, but merely tells the system to check the load balance after all the ranks call `AMPI_Migrate`. To migrate the rank or not is decided only by the system's load balancing strategy.

The AMPI runtime system could detect load imbalance by itself and invoke the load balancing strategy. However, if the application code is going to pack/unpack the rank's data, writing the pack subroutine will be complicated if migrations occur at a stage unknown to the application. For example, if the system decides to migrate a rank while it is in initialization stage (say, reading input files), application code will have to keep track of how much data it has read, what files are open etc. Typically, since initialization occurs only once in the beginning, load imbalance at that stage would not matter much. Therefore, we want the demand to perform a load balance check to be initiated by the application.

Essentially, a call to `AMPI_Migrate` signifies to the runtime system that the application has reached a point at which it is safe to serialize the local state. Knowing this, the runtime system can act in several ways.

The `MPI_Info` object taken as a parameter by `AMPI_Migrate` gives users a way to influence the runtime system's decision-making and behavior. AMPI provides two built-in `MPI_Info` objects for this, called `AMPI_INFO_LB_SYNC` and `AMPI_INFO_LB_ASYNC`. Synchronous load balancing assumes that the application is already at a synchronization point. Asynchronous load balancing does not assume this.

Calling `AMPI_Migrate` on a rank with pending send requests (i.e. from `MPI_Isend`) is currently not supported, therefore users should always wait on any outstanding send requests before calling `AMPI_Migrate`.

```

// Main time-stepping loop
for (int iter=0; iter < max_iters; iter++) {

```

(continues on next page)

(continued from previous page)

```

// Time step work ...

if (iter % lb_freq == 0)
    AMPI_Migrate(AMPI_INFO_LB_SYNC);
}

```

Note that migrating ranks around the cores and nodes of a system can change which ranks share physical resources, such as memory. A consequence of this is that communicators created via `MPI_Comm_split_type` are invalidated by calls to `AMPI_Migrate` that result in migration which breaks the semantics of that communicator type. The only valid routine to call on such communicators is `MPI_Comm_free`.

We also provide callbacks that user code can register with the runtime system to be invoked just before and right after migration: `AMPI_Register_about_to_migrate` and `AMPI_Register_just_migrated` respectively. Note that the callbacks are only invoked on those ranks that are about to actually migrate or have just actually migrated.

AMPI provide routines for starting and stopping load measurements, and for users to explicitly set the load value of a rank using the following: `AMPI_Load_start_measure`, `AMPI_Load_stop_measure`, `AMPI_Load_reset_measure`, and `AMPI_Load_set_value`. And since AMPI builds on top of Charm++, users can experiment with the suite of load balancing strategies included with Charm++, as well as write their own strategies based on user-level information and heuristics.

3.4.3 Checkpointing and Fault Tolerance

Using the same serialization functionality as AMPI's migration support, it is also possible to save the state of the program to disk, so that if the program were to crash abruptly, or if the allocated time for the program expires before completing execution, the program can be restarted from the previously checkpointed state.

To perform a checkpoint in an AMPI program, all you have to do is make a call to `int AMPI_Migrate(MPI_Info hints)` with an `MPI_Info` object that specifies how you would like to checkpoint. Checkpointing can be thought of as migrating AMPI ranks to storage. Users set the checkpointing policy on an `MPI_Info` object's "ampi_checkpoint" key to one of the following values: "to_file=directory_name" or "false". To perform checkpointing in memory a built-in `MPI_Info` object called `AMPI_INFO_CHKPT_IN_MEMORY` is provided.

Checkpointing to file tells the runtime system to save checkpoints in a given directory. (Typically, in an iterative program, the iteration number, converted to a character string, can serve as a checkpoint directory name.) This directory is created, and the entire state of the program is checkpointed to this directory. One can restart the program from the checkpointed state (using the same, more, or fewer physical processors than were checkpointed with) by specifying "+restart directory_name" on the command-line.

Checkpointing in memory allows applications to transparently tolerate failures online. The checkpointing scheme used here is a double in-memory checkpoint, in which virtual processors exchange checkpoints pairwise across nodes in each other's memory such that if one node fails, that failed node's AMPI ranks can be restarted by its buddy once the failure is detected by the runtime system. As long as no two buddy nodes fail in the same checkpointing interval, the system can restart online without intervention from the user (provided the job scheduler does not revoke its allocation). Any load imbalance resulting from the restart can then be managed by the runtime system. Use of this scheme is illustrated in the code snippet below.

```

// Main time-stepping loop
for (int iter=0; iter < max_iters; iter++) {

    // Time step work ...

    if (iter % chkpt_freq == 0)

```

(continues on next page)

(continued from previous page)

```

    AMPI_Migrate(AMPI_INFO_CHKPT_IN_MEMORY);
}

```

A value of "false" results in no checkpoint being done that step. Note that `AMPI_Migrate` is a collective function, meaning every virtual processor in the program needs to call this subroutine with the same `MPI_Info` object. The checkpointing capabilities of AMPI are powered by the Charm++ runtime system. For more information about checkpoint/restart mechanisms please refer to the Charm++ manual: [2.3.12](#).

3.4.4 Memory Efficiency

MPI functions usually require the user to preallocate the data buffers needed before the functions being called. For unblocking communication primitives, sometimes the user would like to do lazy memory allocation until the data actually arrives, which gives the opportunities to write more memory efficient programs. We provide a set of AMPI functions as an extension to the standard MPI-2 one-sided calls, where we provide a split phase `MPI_Get` called `AMPI_Iget`. `AMPI_Iget` preserves the similar semantics as `MPI_Get` except that no user buffer is provided to hold incoming data. `AMPI_Iget_wait` will block until the requested data arrives and runtime system takes care to allocate space, do appropriate unpacking based on data type, and return. `AMPI_Iget_free` lets the runtime system free the resources being used for this get request including the data buffer. Finally, `AMPI_Iget_data` is the routine used to access the data.

```

int AMPI_Iget(MPI_Aint orgdisp, int orgcnt, MPI_Datatype orgtype, int rank,
             MPI_Aint targdisp, int targcnt, MPI_Datatype targtype, MPI_Win win,
             MPI_Request *request);

int AMPI_Iget_wait(MPI_Request *request, MPI_Status *status, MPI_Win win);

int AMPI_Iget_free(MPI_Request *request, MPI_Status *status, MPI_Win win);

int AMPI_Iget_data(void *data, MPI_Status status);

```

3.4.5 Compute Resource Awareness

AMPI provides a set of built-in attributes on all communicators to find the number of the worker thread or process that a rank is currently running on, its home worker thread, as well as the total number of worker threads and processes in the job. We define a worker thread to be a thread on which one or more AMPI ranks are scheduled. We define a process here as an operating system process, which may contain one or more worker threads. The built-in attributes are listed in the following table:

Attribute	Defintion
<code>AMPI_MY_WTH</code>	Worker thread the rank is currently running on.
<code>AMPI_MY_PROCESS</code>	OS process the rank is currently running on.
<code>AMPI_NUM_WTHS</code>	Number of worker threads in the application.
<code>AMPI_NUM_PROCESSES</code>	Number of OS processes in the application.
<code>AMPI_MY_HOME_WTH</code>	Home worker thread of the rank.

These attributes are accessible from any rank by calling `MPI_Comm_get_attr`, such as:

```

! Fortran:
integer (kind=MPI_ADDRESS_KIND) :: my_wth_ptr
integer :: my_wth, flag, ierr

```

(continues on next page)

(continued from previous page)

```
call MPI_Comm_get_attr(MPI_COMM_WORLD, AMPI_MY_WTH, my_wth_ptr, flag, ierr)
my_wth = my_wth_ptr
```

```
// C/C++:
int * my_wth_ptr;
int my_wth, flag;
MPI_Comm_get_attr(MPI_COMM_WORLD, AMPI_MY_WTH, &my_wth_ptr, &flag);
my_wth = *my_wth_ptr;
```

Warning: The pointers retrieved for these attributes will become invalid after migration. Always copy their values into local variables if you need to access the old values after a migration.

AMPI also provides extra communicator types that users can pass to `MPI_Comm_split_type`: `AMPI_COMM_TYPE_HOST` for splitting a communicator into disjoint sets of ranks that share the same physical host, `AMPI_COMM_TYPE_PROCESS` for splitting a communicator into disjoint sets of ranks that share the same operating system process, and `AMPI_COMM_TYPE_WTH`, for splitting a communicator into disjoint sets of ranks that share the same worker thread.

3.4.6 Charm++ Interoperation

There is preliminary support for interoperating AMPI programs with Charm++ programs. This allows users to launch an AMPI program with an arbitrary number of virtual processes in the same executable as a Charm++ program that contains arbitrary collections of chares, with both AMPI ranks and chares being co-scheduled by the runtime system. We also provide an entry method `void injectMsg(int n, char buf[n])` for chares to communicate with AMPI ranks. An example program can be found in `examples/charm++/AMPI-interop`.

3.4.7 Sequential Re-run of a Parallel Node

In some scenarios, a sequential re-run of a parallel node is desired. One example is instruction-level accurate architecture simulations, in which case the user may wish to repeat the execution of a node in a parallel run in the sequential simulator. AMPI provides support for such needs by logging the change in the MPI environment on a certain processors. To activate the feature, build AMPI module with variable “AMPIMSGLOG” defined, like the following command in charm directory. (Linking with zlib “-lz” might be required with this, for generating compressed log file.)

```
$ ./build AMPI netlrts-linux-x86_64 -DAMPIMSGLOG
```

The feature is used in two phases: writing (logging) the environment and repeating the run. The first logging phase is invoked by a parallel run of the AMPI program with some additional command line options.

```
$ ./charmrun ./pgm +p4 +vp4 +msgLogWrite +msgLogRank 2 +msgLogFilename "msg2.log"
```

In the above example, a parallel run with 4 worker threads and 4 AMPI ranks will be executed, and the changes in the MPI environment of worker thread 2 (also rank 2, starting from 0) will get logged into diskfile “msg2.log”.

Unlike the first run, the re-run is a sequential program, so it is not invoked by `charmrun` (and omitting `charmrun` options like `+p4` and `+vp4`), and additional command line options are required as well.

```
$ ./pgm +msgLogRead +msgLogRank 2 +msgLogFilename "msg2.log"
```

3.4.8 User Defined Initial Mapping

By default AMPI maps virtual processes to processing elements in a blocked fashion. This maximizes communication locality in the common case, but may not be ideal for all applications. With AMPI, users can define the initial mapping of virtual processors to physical processors at runtime, either choosing from the predefined initial mappings below or defining their own mapping in a file.

Round Robin This mapping scheme maps virtual processor to physical processor in round-robin fashion, i.e. if there are 8 virtual processors and 2 physical processors then virtual processors indexed 0,2,4,6 will be mapped to physical processor 0 and virtual processors indexed 1,3,5,7 will be mapped to physical processor 1.

```
$ ./charmrun ./hello +p2 +vp8 +mapping RR_MAP
```

Block Mapping This mapping scheme maps virtual processors to physical processor in ranks, i.e. if there are 8 virtual processors and 2 physical processors then virtual processors indexed 0,1,2,3 will be mapped to physical processor 0 and virtual processors indexed 4,5,6,7 will be mapped to physical processor 1.

```
$ ./charmrun ./hello +p2 +vp8 +mapping BLOCK_MAP
```

Proportional Mapping This scheme takes the processing capability of physical processors into account for mapping virtual processors to physical processors, i.e. if there are 2 processors running at different frequencies, then the number of virtual processors mapped to processors will be in proportion to their processing power. To make the load balancing framework aware of the heterogeneity of the system, the flag *+LBTestPESpeed* should also be used.

```
$ ./charmrun ./hello +p2 +vp8 +mapping PROP_MAP
$ ./charmrun ./hello +p2 +vp8 +mapping PROP_MAP +balancer GreedyLB +LBTestPESpeed
```

Custom Mapping To define your own mapping scheme, create a file named “mapfile” which contains on each line the PE number you’d like that virtual process to start on. This file is read when specifying the *+mapping MAPFILE* option. The following mapfile will result in VPs 0, 2, 4, and 6 being created on PE 0 and VPs 1, 3, 5, and 7 being created on PE 1:

```
0
1
0
1
0
1
0
1
```

```
$ ./charmrun ./hello +p2 +vp8 +mapping MAPFILE
```

Note that users can find the current mapping of ranks to PEs (after dynamic load balancing) by calling `AMPI_Comm_get_attr` on `MPI_COMM_WORLD` with the predefined `AMPI_MY_WTH` attribute. This information can be gathered and dumped to a file for use in future runs as the mapfile.

3.4.9 Performance Visualization

AMPI users can take advantage of Charm++’s tracing framework and associated performance visualization tool, Projections. Projections provides a number of different views of performance data that help users diagnose performance issues. Along with the traditional Timeline view, Projections also offers visualizations of load imbalance and communication-related data.

In order to generate tracing logs from an application to view in Projections, link with `ampicc -tracemode projections`.

AMPI defines the following extensions for tracing support:

AMPI_Trace_begin	AMPI_Trace_end
------------------	----------------

When using the *Timeline* view in Projections, AMPI users can visualize what each VP on each processor is doing (what MPI method it is running or blocked in) by clicking the *View* tab and then selecting *Show Nested Bracketed User Events* from the drop down menu. See the Projections manual for information on performance analysis and visualization.

AMPI users can also use any tracing libraries or tools that rely on MPI's PMPI profiling interface, though such tools may not be aware of AMPI process virtualization.

3.5 Example Applications

This section contains a list of applications that have been written or adapted to work with AMPI. Most applications are available on git:

```
git clone ssh://charm.cs.illinois.edu:9418/benchmarks/ampi-benchmarks.
```

Most benchmarks can be compiled with the provided top-level Makefile:

```
$ git clone ssh://charm.cs.illinois.edu:9418/benchmarks/ampi-benchmarks
$ cd ampi-benchmarks
$ make -f Makefile.ampi
```

3.5.1 Mantevo project v3.0

Set of mini-apps from the Mantevo project. Download at <https://mantevo.org/download/>.

MiniFE

- Mantevo mini-app for unstructured implicit Finite Element computations.
- No changes necessary to source to run on AMPI. Modify file `makefile.ampi` and change variable `AMPIDIR` to point to your Charm++ directory, execute `make -f makefile.ampi` to build the program.
- Refer to the README file on how to run the program. For example: `./charmrun +p4 ./miniFE.x nx=30 ny=30 nz=30 +vp32`

MiniMD v2.0

- Mantevo mini-app for particle interaction in a Lennard-Jones system, as in the LAMMPS MD code.
- No changes necessary to source code. Modify file `Makefile.ampi` and change variable `AMPIDIR` to point to your Charm++ directory, execute `make ampi` to build the program.
- Refer to the README file on how to run the program. For example: `./charmrun +p4 ./miniMD_ampi +vp32`

CoMD v1.1

- Mantevo mini-app for molecular dynamics codes: <https://github.com/exmatex/CoMD>
- To AMPI-ize it, we had to remove calls to not thread-safe `getopt()`. Support for dynamic load balancing has been added in the main loop and the command line options. It will run on all platforms.
- Just update the Makefile to point to AMPI compilers and run with the provided run scripts.

MiniXYCE v1.0

- Mantevo mini-app for discrete analog circuit simulation, version 1.0, with serial, MPI, OpenMP, and MPI+OpenMP versions.
- No changes besides Makefile necessary to run with virtualization. To build, do `cp common/generate_info_header miniXyce_ref/.`, modify the CC path in `miniXyce_ref/` and run `make`. Run scripts are in `test/`.
- Example run command: `./charmrun +p3 ./miniXyce.x +vp3 -circuit ../tests/cir1.net -t_start 1e-6 -pf params.txt`

HPCCG v1.0

- Mantevo mini-app for sparse iterative solves using the Conjugate Gradient method for a problem similar to that of MiniFE.
- No changes necessary except to set compilers in Makefile to the AMPI compilers.
- Run with a command such as: `./charmrun +p2 ./test_HPCCG 20 30 10 +vp16`

MiniAMR v1.0

- miniAMR applies a stencil calculation on a unit cube computational domain, which is refined over time.
- No changes if using swap-globals. Explicitly extern global variables if using TLS.

Not yet AMPI-zed (reason)

MiniAero v1.0 (build issues), MiniGhost v1.0.1 (globals), MiniSMAC2D v2.0 (globals), TeaLeaf v1.0 (globals), CloverLeaf v1.1 (globals), CloverLeaf3D v1.0 (globals).

3.5.2 LLNL ASC Proxy Apps

LULESH v2.0

- LLNL Unstructured Lagrangian-Eulerian Shock Hydrodynamics proxy app: <https://codesign.llnl.gov/lulesh.php>
- Charm++, MPI, MPI+OpenMP, Liszt, Loci, Chapel versions all exist for comparison.
- Manually privatized version of LULESH 2.0, plus a version with PUP routines in subdirectory `pup_lulesh202/`.

AMG 2013

- LLNL ASC proxy app: Algebraic Multi-Grid solver for linear systems arising from unstructured meshes: <https://codesign.llnl.gov/amg2013.php>
- AMG is based on HYPRE, both from LLNL. The only change necessary to get AMG running on AMPI with virtualization is to remove calls to HYPRE's timing interface, which is not thread-safe.
- To build, point the CC variable in Makefile.include to your AMPI CC wrapper script and make. Executable is test/amg2013.

Lassen v1.0

- LLNL ASC mini-app for wave-tracking applications with dynamic load imbalance. Reference versions are serial, MPI, Charm++, and MPI/Charm++ interop: <https://codesign.llnl.gov/lassen.php>
- No changes necessary to enable AMPI virtualization. Requires some C++11 support. Set AMPIDIR in Makefile and make. Run with: `./charmrun +p4 ./lassen_mpi +vp8 default 2 2 2 50 50 50`

Kripke v1.1

- LLNL ASC proxy app for ARDRA, a full Sn deterministic particle transport application: <https://codesign.llnl.gov/kripke.php>
- Charm++, MPI, MPI+OpenMP, MPI+RAJA, MPI+CUDA, MPI+OCCA versions exist for comparison.
- Kripke requires no changes between MPI and AMPI since it has no global/static variables. It uses cmake so edit the cmake toolchain files in cmake/toolchain/ to point to the AMPI compilers, and build in a build directory:

```
$ mkdir build; cd build;
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=../cmake/Toolchain/linux-gcc-mpi.cmake
-DENABLE_OPENMP=OFF
$ make
```

Run with:

```
$ ./charmrun +p8 ./src/tools/kripke +vp8 --zones 64,64,64 --procs 2,2,2 --nest ZDG
```

MCB v1.0.3 (2013)

- LLNL ASC proxy app for Monte Carlo particle transport codes: <https://codesign.llnl.gov/mcb.php>
- MPI+OpenMP reference version.
- Run with:

```
$ OMP_NUM_THREADS=1 ./charmrun +p4 ../../src/MCBenchmark.exe --weakScaling
--distributedSource --nCores=1 --numParticles=20000 --multiSigma --nThreadCore=1
↪+vp16
```

Not yet AMPI-zed (reason)

: UMT 2013 (global variables).

3.5.3 Other Applications

MILC 7.0

- MILC is a code to study quantum chromodynamics (QCD) physics. <http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/milc/>
- Moved `MPI_Init_thread` call to `main()`, added `__thread` to all global/static variable declarations. Runs on AMPI with virtualization when using `-tsglobals`.
- Build: edit `ks_imp_ds/Makefile` to use AMPI compiler wrappers, run `make su3_rmd` in `ks_imp_ds/`
- Run with: `./su3_rmd +vp8 ../benchmark_n8/single_node/n8_single.in`

SNAP v1.01 (C version)

- LANL proxy app for PARTISN, an Sn deterministic particle transport application: <https://github.com/losalamos/SNAP>
- SNAP is an update to Sweep3D. It simulates the same thing as Kripke, but with a different decomposition and slight algorithmic differences. It uses a 1- or 2-dimensional decomposition and the KBA algorithm to perform parallel sweeps over the 3-dimensional problem space. It contains all of the memory, computation, and network performance characteristics of a real particle transport code.
- Original SNAP code is Fortran90-MPI-OpenMP, but this is a C-MPI-OpenMP version of it provided along with the original version. The Fortran90 version will require global variable privatization, while the C version works out of the box on all platforms.
- Edit the Makefile for AMPI compiler paths and run with: `./charmrun +p4 ./snap +vp4 --fi center_src/fin01 --fo center_src/fout01`

Sweep3D

- Sweep3D is a *particle transport* program that analyzes the flux of particles along a space. It solves a three-dimensional particle transport problem.
- This mini-app has been deprecated, and replaced at LANL by SNAP (above).
- Build/Run Instructions:
 - Modify the `makefile` and change variable `CHARMC` to point to your Charm++ compiler command, execute `make mpi` to build the program.
 - Modify file `input` to set the different parameters. Refer to file `README` on how to change those parameters. Run with: `./charmrun ./sweep3d.mpi +p8 +vp16`

PENNANT v0.8

- Unstructured mesh Rad-Hydro mini-app for a full application at LANL called FLAG. <https://github.com/losalamos/PENNANT>
- Written in C++, only global/static variables that need to be privatized are `mype` and `numpe`. Done manually.
- Legion, Regent, MPI, MPI+OpenMP, MPI+CUDA versions of PENNANT exist for comparison.
- For PENNANT-v0.8, point `CC` in `Makefile` to `AMPICC` and just 'make'. Run with the provided input files, such as: `./charmrun +p2 ./build/pennant +vp8 test/noh/noh.pnt`

3.5.4 Benchmarks

Jacobi-2D (Fortran)

- Jacobi-2D with 1D decomposition. Problem size and number of iterations are defined in the source code. Manually privatized.

Jacobi-3D (C)

- Jacobi-3D with 3D decomposition. Manually privatized. Includes multiple versions: Isomalloc, PUP, FT, LB, Isend/Irecv, Iput/Iget.

NAS Parallel Benchmarks (NPB 3.3)

- A collection of kernels used in different scientific applications. They are mainly implementations of various linear algebra methods. <http://www.nas.nasa.gov/Resources/Software/npb.html>
- Build/Run Instructions:
 - Modify file `config/make.def` to make variable `CHAMRDIR` point to the right Charm++ directory.
 - Use `make <benchmark> NPROCS=<P> CLASS=<C>` to build a particular benchmark. The values for `<benchmark>` are (bt, cg, dt, ep, ft, is, lu, mg, sp), `<P>` is the number of ranks and `<C>` is the class or the problem size (to be chosen from A,B,C,D or E). Some benchmarks may have restrictions on values of `<P>` and `<C>`. For instance, to make CG benchmark with 256 ranks and class C, we will use the following command: `make cg NPROCS=256`
 - The resulting executable file will be generated in the respective directory for the benchmark. In the previous example, a file `cg.256.C` will appear in the `CG` and `bin/` directories. To run the particular benchmark, you must follow the standard procedure of running AMPI programs: `./charmrun ./cg.C.256 +p64 +vp256 ++nodelist nodelist`

NAS PB Multi-Zone Version (NPB-MZ 3.3)

- A multi-zone version of BT, SP and LU NPB benchmarks. The multi-zone intentionally divides the space unevenly among ranks and causes load imbalance. The original goal of multi-zone versions was to offer an test case for hybrid MPI+OpenMP programming, where the load imbalance can be dealt with by increasing the number of threads in those ranks with more computation. <http://www.nas.nasa.gov/Resources/Software/npb.html>
- The BT-MZ program shows the heaviest load imbalance.
- Build/Run Instructions:
 - Modify file `config/make.def` to make variable `CHAMRDIR` point to the right Charm++ build.
 - Use the format `make <benchmark> NPROCS=<P> CLASS=<C>` to build a particular benchmark. The values for `<benchmark>` are (bt-mz, lu-mz, sp-mz), `<P>` is the number of ranks and `<C>` is the class or the problem size (to be chosen from A,B,C,D or E). Some benchmarks may have restrictions on values of `<P>` and `<C>`. For instance, to make the BT-MZ benchmark with 256 ranks and class C, you can use the following command: `make bt-mz NPROCS=256 CLASS=C`
 - The resulting executable file will be generated in the `bin/` directory. In the previous example, a file `bt-mz.256.C` will be created in the `bin` directory. To run the particular benchmark, you must follow the standard procedure of running AMPI programs: `./charmrun ./bt-mz.C.256 +p64 +vp256 ++nodelist nodelist`

HPCG v3.0

- High Performance Conjugate Gradient benchmark, version 3.0. Companion metric to Linpack, with many vendor-optimized implementations available: <http://hpcg-benchmark.org/>
- No AMPI-ization needed. To build, modify `setup/Make.AMPI` for compiler paths, do `mkdir build && cd build && configure ../setup/Make.AMPI && make`. To run, do `./charmrun +p16 ./bin/xhpcg +vp64`

Intel Parallel Research Kernels (PRK) v2.16

- A variety of kernels (Branch, DGEMM, Nstream, Random, Reduce, Sparse, Stencil, Synch_global, Synch_p2p, and Transpose) implemented for a variety of runtimes (SERIAL, OpenMP, MPI-1, MPI-RMA, MPI-SHM, MPI+OpenMP, SHMEM, FG_MPI, UPC, Grappa, Charm++, and AMPI). <https://github.com/ParRes/Kernels>
- For AMPI tests, set `CHARMTOP` and run: `make allampi`. There are run scripts included.

OSU Microbenchmarks

MPI collectives performance testing suite. <https://charm.cs.illinois.edu/gerrit/#/admin/projects/benchmarks/osu-collectives-benchmarking>

- Build with: `./configure CC=~charm/bin/ampicc && make`

3.5.5 Third Party Open Source Libraries

HYPRE-2.11.1

- High Performance Preconditioners and solvers library from LLNL. https://computation.llnl.gov/project/linear_solvers/software.php
- Hypre-2.11.1 builds on top of AMPI using the configure command:

```
$ ./configure --with-MPI \
    CC=~charm/bin/ampicc \
    CXX=~charm/bin/ampicxx \
    F77=~charm/bin/ampif77 \
    --with-MPI-include=~charm/include \
    --with-MPI-lib-dirs=~charm/lib \
    --with-MPI-libs=mpi --without-timing --without-print-errors
$ make -j8
```

- All HYPRE tests and examples pass tests with virtualization, migration, etc. except for those that use Hypre's timing interface, which uses a global variable internally. So just remove those calls and do not define `HYPRE_TIMING` when compiling a code that uses Hypre. In the examples directory, you'll have to set the compilers to your AMPI compilers explicitly too. In the test directory, you'll have to edit the Makefile to 1) Remove `-DHYPRE_TIMING` from both `CDEFS` and `CXXDEFS`, 2) Remove both `#{MPILIBS}` and `#{MPIFLAGS}` from `MPILIBFLAGS`, and 3) Remove `#{LIBS}` from `LIBFLAGS`. Then run `make`.
- To run the `new_ij` test, run: `./charmrun +p64 ./new_ij -n 128 128 128 -P 4 4 4 -intertype 6 -tol 1e-8 -CF 0 -solver 61 -agg_nl 1 27pt -Pmx 6 -ns 4 -mu 1 -hmis -rlx 13 +vp64`

MFEM-3.2

- MFEM is a scalable library for Finite Element Methods developed at LLNL. <http://mfem.org/>
- MFEM-3.2 builds on top of AMPI (and METIS-4.0.3 and HYPRE-2.11.1). Download MFEM, [HYPRE](#), and [METIS](#). Untar all 3 in the same top-level directory.
- Build HYPRE-2.11.1 as described above.
- Build METIS-4.0.3 by doing `cd metis-4.0.3/ && make`
- Build MFEM-3.2 serial first by doing `make serial`
- Build MFEM-3.2 parallel by doing:
 - First, comment out `#define HYPRE_TIMING` in `mfem/linalg/hypre.hpp`. Also, you must add a `#define hypre_clearTiming()` at the top of `linalg/hypre.cpp`, because Hype-2.11.1 has a bug where it doesn't provide a definition of this function if you don't define `HYPRE_TIMING`.
 - `make parallel MFEM_USE_MPI=YES MPICXX=~/.charm/bin/ampicxx HYPRE_DIR=~/.hypre-2.11.1/src/hypre METIS_DIR=~/.metis-4.0.3`
- To run an example, do `./charmrun +p4 ./ex15p -m ../data/amr-quad.mesh +vp16`. You may want to add the runtime options `-no-vis` and `-no-visit` to speed things up.
- All example programs and miniapps pass with virtualization, and migration if added.

XBraid-1.1

- XBraid is a scalable library for parallel time integration using MultiGrid, developed at LLNL. <https://computation.llnl.gov/project/parallel-time-integration/software.php>
- XBraid-1.1 builds on top of AMPI (and its examples/drivers build on top of MFEM-3.2, HYPRE-2.11.1, and METIS-4.0.3 or METIS-5.1.0).
- To build XBraid, modify the variables `CC`, `MPICC`, and `MPICXX` in `makefile.inc` to point to your AMPI compilers, then do `make`.
- To build XBraid's examples/ and drivers/ modify the paths to MFEM and HYPRE in their Makefiles and `make`.
- To run an example, do `./charmrun +p2 ./ex-02 -pgrid 1 1 8 -ml 15 -nt 128 -nx 33 33 -mi 100 +vp8 ++local`.
- To run a driver, do `./charmrun +p4 ./drive-03 -pgrid 2 2 2 2 -nl 32 32 32 -nt 16 -ml 15 +vp16 ++local`

3.5.6 Other AMPI codes

- FLASH
- BRAMS (Weather prediction model)
- CGPOP
- Fractography3D (Crack Propagation)
- JetAlloc
- PlasComCM (XPACC)
- PlasCom2 (XPACC)
- Harm3D

Fortran90 Bindings for Charm++

Contents

- *Fortran90 Bindings for Charm++*
 - *Overview*
 - *Writing the Charm++ Interface File*
 - *Writing the F90 Program*
 - *Compilation and Linking*
 - *Running the Program*

Charm++ is a parallel object language based on C++. The `f90charm` module is to provide Fortran90 programs a `f90` interface to Charm++. Using the `F90Charm` interface, users can write Fortran90 programs in a fashion similar to Charm++, which allows creation of parallel object arrays (Chare Arrays) and sending messages between them.

To interface Fortran90 to Charm++ and thus obtain a parallel version of your program you need to do the following things:

1. Write a Charm Interface file (extension `.ci`)
2. Write your F90 program with `f90charmmain()` as main program;
3. Write implementations of Chare entry methods in the F90 program;
4. Compile and Link with Charm's Fortran library
5. Run it!

4.1 Overview

Here we suppose you already know most concepts in Charm++ and have done some Charm++ programming. Unlike in C++, we don't have classes in Fortran90. Thus, a Chare in `F90Charm` is represented as a Fortran type structure.

Here is an example:

```
! ## Just replace Hello throughout with your chare's name. ##
! ## and add your chare's personal data below where indicated ##
! ## Everything else remains the same ##
MODULE HelloMod

TYPE Hello
! ## your chare's data goes here, the integer below is an example ##
integer data
END TYPE

TYPE HelloPtr
TYPE (Hello), POINTER :: obj
integer*8 aid
END TYPE

END MODULE
```

You can think of this module as a Chare declaration. Type [Hello] defines arbitrary user program data and HelloPtr defines the Chare pointer which the Fortran program will use later to communicate with the F90Charm runtime library. The [aid] is the handle of the array returned by the F90Charm library, and the user shouldn't change it.

In F90Charm as in Charm++, you need to write a .ci interface file so that the Charm translator will generate helper functions. The syntax of .ci files is the same as in Charm++, however, for F90Charm, there are certain constraints. First, you don't need to declare the main chare as in Charm++. Second, F90Charm currently only supports up to 3D Chare arrays, and you cannot define Chare, Group, and NodeGroup types. Third, there are no message declarations in .ci files, all the entry functions must be declared in the parameter marshalling fashion of Charm++. Essentially, users can declare in .ci files readonly variables and 1-3D chare arrays with parameter marshalled entry methods.

It is the programmer's responsibility to write the implementation of Chare entry methods. The decl and def files generated by Charm++'s translator define the interface functions programmer need to write.

For each Chare defined in the .ci file, the user must write these functions for F90Charm's runtime:

```
SUBROUTINE <ChareName>_allocate(objPtr, aid, index)
```

You can think of this function as a constructor for each array element with array index [index]. For a 3D array, for example, you can replace index in the example by a 3D array index [index1, index2, index3]. In this function the user must allocate memory for the Chare's user data and perform any initialization.

For each Chare entry method you have declared, you should write the corresponding Fortran90 subroutine for it:

```
SUBROUTINE <ChareName>_Entry_<EntryName>(charePtr, myIndex, data1, data2 ... )
```

Note that the first argument is the Chare pointer as declared previously, and the second argument is the array index which will be passed from the Charm runtime. The rest of the parameters should be the same as declared in the .ci file. For higher dimensional arrays, replace myIndex by myIndex1, myIndex2 for example.

On the caller side, the decl/def files generated by Charm++'s translator also provide the following functions for Chare creation and remote method invocation. For each Chare declared in .ci files, these subroutines are generated for use in Fortran90 code:

```
<ChareName>_CkNew(integer n, integer*8 aid)
```

This subroutine creates a chare array of size n. For higher dimensional array creation, specify one integer for each dimension. For example, to create a 3D array:

```
<ChareName>_CkNew(integer dim1, integer dim2, integer dim3, integer*8 aid)
```

For each entry method, a corresponding Invoke method is provided (1D example shown here):

```
<ChareName>_Invoke_<EntryName>(charePtr, myIndex, data1, data2 ... )
```

This subroutine will send a message to the array element with index `myIndex`. Similarly, for arrays with higher dimensions, replace `myIndex` by a corresponding number of array indices. Broadcasts are supported as well:

```
<ChareName>_Broadcast_<EntryName>(charePtr, data1, data2 ... )
```

There are several others things you need to know.

First, as in Charm++, each `.ci` file will generate two header files: `.decl.h` and `.def.h`. However, in Fortran90 charm, you are not able to include these C++ files in Fortran90 code. Thus, currently, it is the user's task to write a short C++ file including these two headers files. You should also provide definitions for readonly variables in this C++ file. It can be as simple as this:

```
#include "hello.decl.h"
int chunkSize; // define readonly variables here
#include "hello.def.h"
```

In the future, this file could be generated automatically by the translator.

Second, you can still use readonly variables as in Charm++. However, since there are no global variables as in C++ in Fortran90, you have to access them explicitly via function call. Here are the two helper functions that the translator generates:

Take the readonly variable `chunkSize` as an example:

```
Set_Chunksize(chunkSize);
Get_Chunksize(chunkSize);
```

These two functions can be used in user's Fortran program to set and get readonly variables.

Third, for the user's convenience, several Charm++ runtime library functions have their Fortran interface defined in the `F90Charm` library. These currently include:

```
CkExit()
CkMyPe(integer mype)
CkNumPes(integer pes)
CkPrintf(...) // note, the format string must terminated with '$$'
```

Here is a summary of current constraints to write F90 binding Charm++ programs:

1. Only one- to three-dimensional chare arrays are supported.
2. readonly variables must be basic types, i.e. they have to be integers, floats, etc. scalar types or array types of these basic scalar types.
3. Instead of program `main`, your f90 main program starts from subroutine `f90charmmain`.

These details are best illustrated with an example: a hello world program. When executed, an array of several parallel chares is created, forming a ring. Each chare prints a string when it receives a message, and then sends a message to the next chare in the ring. The Fortran `f90charmmain` subroutine begins execution, and the `SayHi` subroutine performs each chare's task.

4.2 Writing the Charm++ Interface File

In this step, you need to write a Charm++ interface file (`.ci`). In the file you can declare parallel chare arrays and their entry methods. The syntax is the same as in Charm++.

```
// ## Just replace Hello throughout with your chare's name. ##
// ## and add your chare's entry points below where indicated ##
// ## Everything else remains the same ##
mainmodule hello {
    // declare readonly variables which once set is available to all
    // Chares across processors.
    readonly int chunkSize;

    array [1D] Hello {
        entry Hello();

        // Note how your Fortran function takes the above defined
        // message instead of a list of parameters.
        entry void SayHi(int a, double b, int n, int arr[n]);

        // Other entry points go here
        entry [reductiontarget] void MyReduction(int result);
    };
};
```

Note, you cannot declare a main chare in the interface file, and you also are not supposed to declare messages. Furthermore, the entry functions must be declared with explicit parameters instead of using messages.

4.3 Writing the F90 Program

To start, you need to create a Fortran Module to represent a chare, e.g. {ChareName}Mod.

```
! ## Just replace Hello throughout with your chare's name. ##
! ## and add your chare's personal data below where indicated ##
! ## Everything else remains the same ##
MODULE HelloMod

TYPE Hello
! ## your chare's data goes here ##
integer data
END TYPE

TYPE HelloPtr
TYPE (Hello), POINTER :: obj
integer*8 aid
END TYPE

END MODULE
```

In the Fortran file you must write an allocate function for this chare with the name: Hello_allocate.

```
! ## Just replace Hello throughout with your chare's name. ##
! ## Everything else remains the same ##
SUBROUTINE Hello_allocate(objPtr, aid, index)
USE HelloMod
TYPE(HelloPtr) objPtr
integer*8 aid
integer index

allocate (objPtr%obj)
```

(continues on next page)

(continued from previous page)

```

objPtr%aid = aid;
! ## you can initialize the Chare user data here
objPtr%obj%data = index
END SUBROUTINE

```

Now that you have the chare and the chare constructor function, you can start to write entry functions as declared in the .ci files.

```

! ## p1, p2, etc represent user parameters
! ## the "objPtr, myIndex" stuff is required in every Entry Point.
! ## CkExit() must be called by the chare to terminate.
SUBROUTINE Hello_Entry_SayHi(objPtr, myIndex, data, data2, len, s)
USE HelloMod
IMPLICIT NONE

TYPE(HelloPtr) objPtr
integer myIndex
integer data
double precision data2
integer len
integer s(len)

objPtr%obj%data = 20
if (myIndex < 4) then
    call Hello_Invoke_SayHi(objPtr%aid, myIndex+1, 1, data2, len, s);
else
    call CkExit()
endif

```

Preliminary support for reductions is available as well. Support is limited to reducing from a chare array to the first member of the same array. Only basic built-in reducers are available. For an entry method named MyReduction, tagged as a reduction target in the interface file, a contribution can be made as follows:

```

external Hello_ReductionTarget_MyReduction

call Hello_contribute(objPtr%aid, myIndex, sizeof(myIndex), myValue, CHARM_SUM_INT,
↳Hello_ReductionTarget_MyReduction)

```

Now, you can write the main program to create the chare array and start the program by sending the first message.

```

SUBROUTINE f90charmmain()
USE HelloMod
integer i
double precision d
integer*8 aid
integer s(8)

call Hello_CkNew(5, aid)

call set_ChunkSize(10);

do i=1,8
    s(i) = i;
enddo
d = 2.50
call Hello_Invoke_SayHi(aid, 0, 1, d, 4, s(3:6));

```

(continues on next page)

(continued from previous page)

END

This main program creates an chare array `Hello` of size 5 and send a message with an integer, an double and array of integers to the array element of index 0.

4.4 Compilation and Linking

Lastly, you need to compile and link the Fortran program with the Charm runtime system as follows: (Let's say you have written `helloof.f90`, `hello.ci` and `hello.C`.)

```
$ charmc hello.ci -language f90charm
```

will create `hello.decl.h` and `hello.def.h`.

```
$ charmc -c hello.C
```

will compile `hello.C` with `hello.decl.h` and `hello.def.h`.

```
$ charmc -c helloof.f90
```

`charmc` will invoke the Fortran compiler:

```
$ charmc -o hello hello.o helloof.o -language f90charm
```

will link `helloof.o` and `hello.o` against Charm's Fortran90 library to create a new executable program, `hello`.

A 2D array example can be found in `charm/examples/charm++/f90charm/hello2D`.

4.5 Running the Program

To run the program, type:

```
$ ./charmrun +p2 hello
```

which will run `hello` on two PEs.

Converse and Charm++ Libraries

Contents

- *Converse and Charm++ Libraries*
 - *Introduction*
 - *liveViz Library*
 - * *Introduction*
 - * *How to use liveViz with Charm++ program*
 - * *Format of deposit image*
 - * *liveViz Initialization*
 - * *Compilation*
 - * *Poll Mode*
 - * *Caveats*
 - *Multi-phase Shared Arrays Library*
 - *3D FFT Library*
 - * *Introduction and Motivation*
 - * *Features*
 - * *Compilation and Execution*
 - * *Library Interface*
 - *TRAM*
 - * *Overview*
 - * *Example*

5.1 Introduction

This manual describes Charm++ and Converse libraries. This is a work in progress towards a standard library for parallel programming on top of the Converse and Charm++ system. All of these libraries are included in the source and binary distributions of Charm++/Converse.

5.2 liveViz Library

5.2.1 Introduction

If array elements compute a small piece of a large 2D image, then these image chunks can be combined across processors to form one large image using the liveViz library. In other words, liveViz provides a way to reduce 2D-image data, which combines small chunks of images deposited by chares into one large image.

This visualization library follows the client server model. The server, a parallel Charm++ program, does all image assembly, and opens a network (CCS) socket which clients use to request and download images. The client is a small Java program. A typical use of this is:

```
cd charm/examples/charm++/wave2d
make
./charmrun ./wave2d +p2 ++server ++server-port 1234
~/ccs_tools/bin/liveViz localhost 1234
```

Use git to obtain a copy of ccs_tools (prior to using liveViz) and build it by:

```
cd ccs_tools;
ant;
```

5.2.2 How to use liveViz with Charm++ program

The liveViz routines are in the Charm++ header “liveViz.h”.

A typical program provides a chare array with one entry method with the following prototype:

```
entry void functionName(liveVizRequestMsg *m);
```

This entry method is supposed to deposit its (array element’s) chunk of the image. This entry method has following structure:

```
void myArray::functionName (liveVizRequestMsg *m)
{
    // prepare image chunk
    ...

    liveVizDeposit (m, startX, startY, width, height, imageBuff, this);

    // delete image buffer if it was dynamically allocated
}
```

Here, “width” and “height” are the size, in pixels, of this array element’s portion of the image, contributed in “imageBuff” (described below). This will show up on the client’s assembled image at 0-based pixel (startX,startY). The client’s display width and height are stored in m->req.wid and m->req.ht.

By default, liveViz combines image chunks by doing a saturating sum of overlapping pixel values. If you want liveViz to combine image chunks by using max (i.e. for overlapping pixels in deposited image chunks, final image will have the pixel with highest intensity or in other words largest value), you need to pass one more parameter (liveVizCombine_t) to the “liveVizDeposit” function:

```
liveVizDeposit (m, startX, startY, width, height, imageBuff, this,
               max_image_data);
```

You can also reduce floating-point image data using sum_float_image_data or max_float_image_data.

5.2.3 Format of deposit image

“imageBuff” is run of bytes representing a rectangular portion of the image. This buffer represents image using a row-major format, so 0-based pixel (x,y) (x increasing to the right, y increasing downward in typical graphics fashion) is stored at array offset “x+y*width”.

If the image is gray-scale (as determined by liveVizConfig, below), each pixel is represented by one byte. If the image is color, each pixel is represented by 3 consecutive bytes representing red, green, and blue intensity.

If the image is floating-point, each pixel is represented by a single ‘float’, and after assembly colored by calling the user-provided routine below. This routine converts fully assembled ‘float’ pixels to RGB 3-byte pixels, and is called only on processor 0 after each client request.

```
extern "C"
void liveVizFloatToRGB(liveVizRequest &req,
    const float *floatSrc, unsigned char *destRgb,
    int nPixels);
```

5.2.4 liveViz Initialization

liveViz library needs to be initialized before it can be used for visualization. For initialization follow the following steps from your main chare:

1. Create your chare array (array proxy object ‘a’) with the entry method ‘functionName’ (described above). You must create the chare array using a CkArrayOptions ‘opts’ parameter. For instance,

```
CkArrayOptions opts(rows, cols);
array = CProxy_Type::ckNew(opts);
```

2. Create a CkCallback object (‘c’), specifying ‘functionName’ as the callback function. This callback will be invoked whenever the client requests a new image.
3. Create a liveVizConfig object (‘cfg’). LiveVizConfig takes a number of parameters, as described below.
4. Call liveVizInit (cfg, a, c, opts).

The liveVizConfig parameters are:

- The first parameter is the pixel type to be reduced:
 - “false” or liveVizConfig::pix_greyscale means a greyscale image (1 byte per pixel).
 - “true” or liveVizConfig::pix_color means a color image (3 RGB bytes per pixel).
 - liveVizConfig::pix_float means a floating-point color image (1 float per pixel, can only be used with sum_float_image_data or max_float_image_data).

- The second parameter is the flag “serverPush”, which is passed to the client application. If set to true, the client will repeatedly request for images. When set to false the client will only request for images when its window is resized and needs to be updated.
- The third parameter is an optional 3D bounding box (type CkBbox3d). If present, this puts the client into a 3D visualization mode.

A typical 2D, RGB, non-push call to liveVizConfig looks like this:

```
liveVizConfig cfg(true, false);
```

5.2.5 Compilation

A Charm++ program that uses liveViz must be linked with ‘-module liveViz’.

Before compiling a liveViz program, the liveViz library may need to be compiled. To compile the liveViz library:

- go to .../charm/tmp/libs/ck-libs/liveViz
- make

5.2.6 Poll Mode

In some cases you may want a server to deposit images only when it is ready to do so. For this case the server will not register a callback function that triggers image generation, but rather the server will deposit an image at its convenience. For example a server may want to create a movie or series of images corresponding to some timesteps in a simulation. The server will have a timestep loop in which an array computes some data for a timestep. At the end of each iteration the server will deposit the image. The use of LiveViz’s Poll Mode supports this type of server generation of images.

Poll Mode contains a few significant differences to the standard mode. First we describe the use of Poll Mode, and then we will describe the differences. liveVizPoll must get control during the creation of your array, so you call liveVizPollInit with no parameters.

```
liveVizPollInit();  
CkArrayOptions opts(nChares);  
arr = CProxy_lvServer::ckNew(opts);
```

To deposit an image, the server just calls liveVizPollDeposit. The server must take care not to generate too many images, before a client requests them. Each server generated image is buffered until the client can get the image. The buffered images will be stored in memory on processor 0.

```
liveVizPollDeposit(this,  
                  startX, startY,           // Location of local piece  
                  localSizeX, localSizeY,   // Dimensions of the piece I'm depositing  
                  globalSizeX, globalSizeY, // Dimensions of the entire image  
                  img,                      // Image byte array  
                  sum_image_data,           // Desired image combiner  
                  3,                        // Bytes/pixel  
                  );
```

The last two parameters are optional. By default they are set to sum_image_data and 3 bytes per pixel.

A sample liveVizPoll server and client are available at:

```
.../charm/examples/charm++/lvServer  
.../ccs_tools/bin/lvClient
```

This example server uses a PythonCCS command to cause an image to be generated by the server. The client also then gets the image.

LiveViz provides multiple image combiner types. Any supported type can be used as a parameter to `liveVizPollDeposit`. Valid combiners include: `sum_float_image_data`, `max_float_image_data`, `sum_image_data`, and `max_image_data`.

The differences in Poll Mode may be apparent. There is no callback function which causes the server to generate and deposit an image. Furthermore, a server may generate an image before or after a client has sent a request. The deposit function, therefore is more complicated, as the server will specify information about the image that it is generating. The client will no longer specify the desired size or other configuration options, since the server may generate the image before the client request is available to the server. The `liveVizPollInit` call takes no parameters.

The server should call `Deposit` with the same global size and combiner type on all of the array elements which correspond to the “this” parameter.

The latest version of `liveVizPoll` is not backwards compatible with older versions. The old version had some fundamental problems which would occur if a server generated an image before a client requested it. Thus the new version buffers server generated images until requested by a client. Furthermore the client requests are also buffered if they arrive before the server generates the images. Problems could also occur during migration with the old version.

5.2.7 Caveats

If you use the old version of “`liveVizInit`” method that only receives 3 parameters, you will find a known bug caused by how “`liveVizDeposit`” internally uses a reduction to build the image.

Using that version of the “`liveVizInit`” method, its contribute call is handled as if it were the chore calling “`liveVizDeposit`” that actually contributed to the `liveViz` reduction. If there is any other reduction going on elsewhere in this chore, some `liveViz` contribute calls might be issued before the corresponding non-`liveViz` contribute is reached. This would imply that image data would be treated as if were part of the non-`liveViz` reduction, leading to unexpected behavior potentially anywhere in the non-`liveViz` code.

5.3 Multi-phase Shared Arrays Library

The Multiphase Shared Arrays (MSA) library provides a specialized shared memory abstraction in Charm++ that provides automatic memory management. Explicitly shared memory provides the convenience of shared memory programming while exposing the performance issues to programmers and the “intelligent” ARTS.

Each MSA is accessed in one specific mode during each phase of execution: `read-only` mode, in which any thread can read any element of the array; `write-once` mode, in which each element of the array is written to (possibly multiple times) by at most one worker thread, and no reads are allowed and `accumulate` mode, in which any threads can add values to any array element, and no reads or writes are permitted. A `sync` call is used to denote the end of a phase.

We permit multiple copies of a page of data on different processors and provide automatic fetching and caching of remote data. For example, initially an array might be put in `write-once` mode while it is populated with data from a file. This determines the cache behavior and the permitted operations on the array during this phase. `write-once` means every thread can write to a different element of the array. The user is responsible for ensuring that two threads do not write to the same element; the system helps by detecting violations. From the cache maintenance viewpoint, each page of the data can be over-written on its owning processor without worrying about transferring ownership or maintaining coherence. At the `sync`, the data is simply merged. Subsequently, the array may be `read-only` for a while, thereafter data might be `accumulate`d into it, followed by it returning to `read-only` mode. In the `accumulate` phase, each local copy of the page on each processor could have its accumulations tracked independently without maintaining page coherence, and the results combined at the end of the phase. The `accumulate` operations also include set-theoretic union operations, i.e. appending items to a set of objects would also be a valid

`accumulate` operation. User-level or compiler-inserted explicit `prefetch` calls can be used to improve performance.

A software engineering benefit that accrues from the explicitly shared memory programming paradigm is the (relative) ease and simplicity of programming. No complex, buggy data-distribution and messaging calculations are required to access data.

To use MSA in a Charm++ program:

- build Charm++ for your architecture, e.g. `netlrts-linux-x86_64`.
- `cd charm/netlrts-linux-x86_64/tmp/libs/ck-libs/multiphaseSharedArrays/;`
`make`
- `#include "msa/msa.h"` in your header file.
- Compile using `charmcc` with the option `-module msa`

The API is as follows: See the example programs in `charm/pgms/charm++/multiphaseSharedArrays`.

5.4 3D FFT Library

The previous 3D FFT library has been deprecated and replaced with this new 3D FFT library. The new 3D FFT library source can be downloaded with following command: `git clone https://charm.cs.illinois.edu/gerrit/libs/fft`

5.4.1 Introduction and Motivation

The 3D Charm-FFT library provides an interface to do parallel 3D FFT computation in a scalable fashion.

The parallelization is achieved by splitting the 3D transform into three phases, using 2D decomposition. First, 1D FFTs are computed over the pencils; then a 'transform' is performed and 1D FFTs are done over second dimension; again a 'transform' is performed and FFTs are computed over the last dimension. So this approach takes three computation phases and two 'transform' phases.

This library allows users to create multiple instances of the library and perform concurrent FFTs using them. Each of the FFT instances run in background as other parts of user code execute, and a callback is invoked when FFT is complete.

5.4.2 Features

Charm-FFT library provides the following features:

- *2D-decomposition*: Users can define fine-grained 2D-decomposition that increases the amount of available parallelism and improves network utilization.
- *Cutoff-based smaller grid*: The data grid may have a cut off. Charm-FFT improves performance by avoiding communication and computation of the data beyond the cutoff.
- *User-defined mapping of library objects*: The placement of objects that constitute the library instance can be defined by the user based on the application's other concurrent communication and placement of other objects.
- *Overlap with other computational work*: Given the callback-based interface and Charm++'s asynchrony, the FFTs are performed in the background while other application work can be done in parallel.

5.4.3 Compilation and Execution

To install the FFT library, you will need to have charm++ installed in your system. You can follow the Charm++ manual to do that. Then, ensure that FFTW3 is installed. FFTW3 can be downloaded from <http://www.fftw.org>. The Charm-FFT library source can be downloaded with following command: `git clone https://charm.cs.illinois.edu/gerrit/libs/fft`

Inside of Charm-FFT directory, you will find *Makefile.default*. Copy this file to *Makefile.common*, change the copy's variable *FFT3_HOME* to point your FFTW3 installation and *CHARM_DIR* to point your Charm++ installation then run *make*. To use Charm-FFT library in an application, add the line *extern module fft_Charm;* to its charm interface (.ci) file and include *fft_charm.h* and *fftw3.h* in relevant C files. Finally to compile the program, pass *-lfft_charm* and *-lfftw3* as arguments to *charmcc*.

5.4.4 Library Interface

To use Charm-FFT interface, the user must start by calling *Charm_createFFT* with following parameters.

```
Charm_createFFT(N_x, N_y, N_z, z_x, z_y, y_x, y_z, x_yz, cutoff, hmati, fft_type, 
↪CkCallback);
```

Where:

```
int N_x : X dimension of FFT calculation
int N_y : Y dimension of FFT calculation
int N_z : Z dimension of FFT calculation
int z_x : X dimension of Z pencil chare array
int z_y : Y dimension of Z pencil chare array
int y_x : X dimension of Y pencil chare array
int y_z : Z dimension of Y pencil chare array
int x_yz : A dimension of X pencil chare array
double cutoff: Cutoff of FFT grid
double *hmati: Hamiltonian matrix representing cutoff
FFT_TYPE: Type of FFT to perform. Either CC for complex-to-complex or RC for real-
↪complex
CkCallback: A Charm++ entry method for callback upon the completion of library 
↪initialization
```

This creates necessary proxies (Z,Y,X etc) for performing FFT of size $N_x \times N_y \times N_z$ using 2D chare arrays (pencils) of size $n_y \times n_x$ (ZPencils), $n_z \times n_x$ (YPencils), and $n_x \times n_y$ (XPencils). When done, calls *myCallback* which should receive *CProxy_fft2d id* as a unique identifier for the newly created set of proxies.

An example of Charm-FFT initialization using *Charm_createFFT*:

```
// .ci
extern module fft_charm;

mainchare Main {
    entry Main(CkArgMsg *m);
}

group Driver {
    entry Driver(FFT_Type fft_type);
    entry void proxyCreated(idMsg *msg);
    entry void fftDone();
}

// .C
Main::Main(CkArgMsg *m) {
```

(continues on next page)

(continued from previous page)

```

...
/* Assume FFT of size N_x, N_y, N_z */
FFT_Type fft_type = CC

    Charm_createFFT(N_x, N_y, N_z, z_x, z_y, y_x, y_z, x_yz, cutoff, hmati,
                  fft_type, CkCallback(CkIndex_Driver::proxyCreated(NULL),
->driverProxy));
}

Driver::proxyCreated(idMsg *msg) {
    CProxy_fft2d fftProxy = msg->id;
    delete msg;
}

```

In this example, an entry method *Driver::proxyCreated* will be called when an FFT instance has been created.

Using the newly received proxy, the user can identify whether a local PE has XPencils and/or ZPencils.

```

void Driver::proxyCreated(idMsg *msg) {
    CProxy_fft2d fftProxy = msg->id;

    delete msg;

    bool hasX = Charm_isOutputPE(fftProxy),
         hasZ = Charm_isInputPE(fftProxy);

    ...
}

```

Then, the grid's dimensions on a PE can be acquired by using *Charm_getOutputExtents* and *Charm_getInputExtents*.

```

if (hasX) {
    Charm_getOutputExtents(gridStart[MY_X], gridEnd[MY_X],
                          gridStart[MY_Y], gridEnd[MY_Y],
                          gridStart[MY_Z], gridEnd[MY_Z],
                          fftProxy);
}

if (hasZ) {
    Charm_getInputExtents(gridStart[MY_X], gridEnd[MY_X],
                         gridStart[MY_Y], gridEnd[MY_Y],
                         gridStart[MY_Z], gridEnd[MY_Z],
                         fftProxy);
}

for(int i = 0; i < 3; i++) {
    gridLength[i] = gridEnd[i] - gridStart[i];
}

```

With the grid's dimension, the user must allocate and set the input and output buffers. In most cases, this is simply the product of the three dimensions, but for real-to-complex FFT calculation, FFTW-style storage for the input buffers is used (as shown below).

```

dataSize = gridLength[MY_X] * gridLength[MY_Y] * gridLength[MY_Z];

if (hasX) {
    dataOut = (complex*) fftw_malloc(dataSize * sizeof(complex));
}

```

(continues on next page)

(continued from previous page)

```

    Charm_setOutputMemory((void*) dataOut, fftProxy);
}

if (hasZ) {
    if (fftType == RC) {
        // FFTW style storage
        dataSize = gridLength[MY_X] * gridLength[MY_Y] * (gridLength[MY_Z]/2 + 1);
    }

    dataIn = (complex*) fftw_malloc(dataSize * sizeof(complex));

    Charm_setInputMemory((void*) dataIn, fftProxy);
}

```

Then, from *PE0*, start the forward or backward FFT, setting the entry method *fftDone* as the callback function that will be called when the FFT operation is complete.

For forward FFT

```

if (CkMyPe() == 0) {
    Charm_doForwardFFT(CkCallback(CkIndex_Driver::fftDone(), thisProxy), fftProxy);
}

```

For backward FFT

```

if (CkMyPe() == 0) {
    Charm_doBackwardFFT(CkCallback(CkIndex_Driver::fftDone(), thisProxy), fftProxy);
}

```

The sample program to run a backward FFT can be found in *Your_Charm_FFT_Path/tests/simple_tests*

5.5 TRAM

5.5.1 Overview

Topological Routing and Aggregation Module is a library for optimization of many-to-many and all-to-all collective communication patterns in Charm++ applications. The library performs topological routing and aggregation of network communication in the context of a virtual grid topology comprising the Charm++ Processing Elements (PEs) in the parallel run. The number of dimensions and their sizes within this topology are specified by the user when initializing an instance of the library.

TRAM is implemented as a Charm++ group, so an *instance* of TRAM has one object on every PE used in the run. We use the term *local instance* to denote a member of the TRAM group on a particular PE.

Most collective communication patterns involve sending linear arrays of a single data type. In order to more efficiently aggregate and process data, TRAM restricts the data sent using the library to a single data type specified by the user through a template parameter when initializing an instance of the library. We use the term *data item* to denote a single object of this datatype submitted to the library for sending. While the library is active (i.e. after initialization and before termination), an arbitrary number of data items can be submitted to the library at each PE.

On systems with an underlying grid or torus network topology, it can be beneficial to configure the virtual topology for TRAM to match the physical topology of the network. This can easily be accomplished using the Charm++ Topology Manager.

The next two sections explain the routing and aggregation techniques used in the library.

Routing

Let the variables j and k denote PEs within an N -dimensional virtual topology of PEs and x denote a dimension of the grid. We represent the coordinates of j and k within the grid as $(j_0, j_1, \dots, j_{N-1})$ and $(k_0, k_1, \dots, k_{N-1})$. Also, let

$$f(x, j, k) = \begin{cases} 0, & \text{if } j_x = k_x \\ 1, & \text{if } j_x \neq k_x \end{cases}$$

j and k are *peers* if

$$\sum_{d=0}^{N-1} f(d, j, k) = 1.$$

When using TRAM, PEs communicate directly only with their peers. Sending to a PE which is not a peer is handled inside the library by routing the data through one or more *intermediate destinations* along the route to the *final destination*.

Suppose a data item destined for PE k is submitted to the library at PE j . If k is a peer of j , the data item will be sent directly to k , possibly along with other data items for which k is the final or intermediate destination. If k is not a peer of j , the data item will be sent to an intermediate destination m along the route to k whose index is $(j_0, j_1, \dots, j_{i-1}, k_i, j_{i+1}, \dots, j_{N-1})$, where i is the greatest value of x for which $f(x, j, k) = 1$.

Note that in obtaining the coordinates of m from j , exactly one of the coordinates of j which differs from the coordinates of k is made to agree with k . It follows that m is a peer of j , and that using this routing process at m and every subsequent intermediate destination along the route eventually leads to the data item being received at k . Consequently, the number of messages $F(j, k)$ that will carry the data item to the destination is

$$F(j, k) = \sum_{d=0}^{N-1} f(d, j, k).$$

Aggregation

Communicating over the network of a parallel machine involves per message bandwidth and processing overhead. TRAM amortizes this overhead by aggregating data items at the source and every intermediate destination along the route to the final destination.

Every local instance of the TRAM group buffers the data items that have been submitted locally or received from another PE for forwarding. Because only peers communicate directly in the virtual grid, it suffices to have a single buffer per PE for every peer. Given a dimension d within the virtual topology, let s_d denote its *size*, or the number of distinct values a coordinate for dimension d can take. Consequently, each local instance allocates up to $s_d - 1$ buffers per dimension, for a total of $\sum_{d=0}^{N-1} (s_d - 1)$ buffers. Note that this is normally significantly less than the total number of PEs specified by the virtual topology, which is equal to $\prod_{d=0}^{N-1} s_d$.

Sending with TRAM is done by submitting a data item and a destination identifier, either PE or array index, using a function call to the local instance. If the index belongs to a peer, the library places the data item in the buffer for the peer's PE. Otherwise, the library calculates the index of the intermediate destination using the previously described algorithm, and places the data item in the buffer for the resulting PE, which by design is always a peer of the local PE. Buffers are sent out immediately when they become full. When a message is received at an intermediate destination, the data items comprising it are distributed into the appropriate buffers for subsequent sending. In the process, if a data item is determined to have reached its final destination, it is immediately delivered.

The total buffering capacity specified by the user may be reached even when no single buffer is completely filled up. In that case the buffer with the greatest number of buffered data items is sent.

Sending to a Chare Array

For sending to a chare array, the entry method should be marked [aggregate], which can take attribute parameters:

```
array [1D] test {
  entry [aggregate(numDimensions: 2, bufferSize: 2048, thresholdFractionNumer : 1,
    thresholdFractionDenom : 2, cutoffFractionNumer : 1,
    cutoffFractionDenom : 2)] void ping(vector<int> data);
};
```

Description of parameters:

- `maxNumDataItemsBuffered`: maximum number of items that the library is allowed to buffer per PE
- `numDimensions`: number of dimensions in grid of PEs
- `bufferSize`: size of the buffer for each peer, in terms of number of data items
- `thresholdFractionNumer`: numerator of the fraction of the buffer that data items
- `thresholdFractionDenom`: size of the buffer for each peer, in terms of number of data items
- `cutoffFractionNumer`: size of the buffer for each peer, in terms of number of data items
- `cutoffFractionDenom`: size of the buffer for each peer, in terms of number of data items

Sending

Sending with TRAM is done through calls to the entry method marked as [aggregate].

Termination

Flushing and termination mechanisms are used in TRAM to prevent deadlock due to indefinite buffering of items. Flushing works by sending out all buffers in a local instance if no items have been submitted or received since the last progress check. Meanwhile, termination detection support is necessary for certain applications.

Currently, the only termination detection method supported is quiescence detection.

When using quiescence detection, no end callback is used, and no done calls are required. Instead, termination of a communication step is achieved using the quiescence detection framework in Charm++, which supports passing a callback as parameter. TRAM is set up such that quiescence will not be detected until all items sent in the current communication step have been delivered to their final destinations.

Periodic flushing is an auxiliary mechanism which checks at a regular interval whether any sends have taken place since the last time the check was performed. If not, the mechanism sends out all the data items buffered per local instance of the library. A typical use case for periodic flushing is when the submission of a data item B to TRAM happens as a result of the delivery of another data item A sent using the same TRAM instance. If A is buffered inside the library and insufficient data items are submitted to cause the buffer holding A to be sent out, a deadlock could arise. With the periodic flushing mechanism, the buffer holding A is guaranteed to be sent out eventually, and deadlock is prevented.

Opting into Fixed-Size Message Handling

Variable-sized message handling in TRAM includes storing and sending additional data that is irrelevant in the case of fixed-size messages. To opt into the faster fixed-size codepath, the `is_PUPbytes` type trait should be explicitly defined for the message type:

```
array [1D] test {  
  entry [aggregate(numDimensions: 2, bufferSize: 2048, thresholdFractionNumer : 1,  
    thresholdFractionDenom : 2, cutoffFractionNumer : 1,  
    cutoffFractionDenom : 2)] void ping(int data);  
};
```

```
template <>  
struct is_PUPbytes<int> {  
  static const bool value = true;  
};
```

5.5.2 Example

For example code showing how to use TRAM, see `examples/charm++/TRAM` and `benchmarks/charm++/streamingAllToAll` in the Charm++ repository.

Frequently Asked Questions

Contents

- *Frequently Asked Questions*
 - *Big Questions*
 - * *What is Charm++?*
 - * *Can Charm++ parallelize my serial program automatically?*
 - * *I can already write parallel applications in MPI. Why should I use Charm++?*
 - * *Will Charm++ run on my machine?*
 - * *Does anybody actually use Charm++?*
 - * *Who created Charm++?*
 - * *What is the future of Charm++?*
 - * *How is Charm++ Licensed?*
 - * *I have a suggestion/feature request/bug report. Who should I send it to?*
 - *Installation and Usage*
 - * *How do I get Charm++?*
 - * *Should I use the GIT version of Charm++?*
 - * *How do I compile Charm++?*
 - * *How do I compile AMPI?*
 - * *Can I remove part of charm tree after compilation to free disk space?*
 - * *If the interactive script fails, how do I compile Charm++?*
 - * *How do I specify the processors I want to use?*

- * *How do I use ssh instead of the deprecated rsh?*
- * *Can I use the serial library X from a Charm program?*
- * *How do I get the command-line switches available for a specific program?*
- * *What should I do if my program hangs while gathering CPU topology information at startup?*
- *Basic Charm++ Programming*
 - * *What's the basic programming model for Charm++?*
 - * *What is an “entry method”?*
 - * *When I invoke a remote method, do I block until that method returns?*
 - * *Why does Charm++ use asynchronous methods?*
 - * *Can I make a method synchronous? Can I then return a value?*
 - * *What is a threaded entry method? How does one make an entry method threaded?*
 - * *If I don't want to use threads, how can an asynchronous method return a value?*
 - * *Isn't there a better way to send data back to whoever called me?*
 - * *Why should I prefer the callback way to return data rather than using [sync] entry methods?*
 - * *How does the initialization in Charm work?*
 - * *Does Charm++ support C and Fortran?*
 - * *What is a proxy?*
 - * *What are the different ways one can create proxies?*
 - * *What is wrong if I do `A *ap = new CProxy_A(handle)`?*
 - * *Why is the `def.h` usually included at the end? Is it necessary or can I just include it at the beginning?*
 - * *How can I use a global variable across different processors?*
 - * *Can I have a class static read-only variable?*
 - * *How do I measure the time taken by a program or operation?*
 - * *What do `CmiAssert` and `CkAssert` do?*
 - * *Can I know how many messages are being sent to a chare?*
 - * *What is “quiescence”? How does it work?*
 - * *Should I use quiescence detection?*
- *Charm++ Arrays*
 - * *How do I know which processor a chare array element is running on?*
 - * *Should I use Charm++ Arrays in my program?*
 - * *Is there a property similar to `thisIndex` containing the chare array's dimensions or total number of elements?*
 - * *How many array elements should I have per processor?*
 - * *What does the term reduction refer to?*
 - * *Can I do multiple reductions on an array?*
 - * *Does Charm++ do automatic load balancing without the user asking for it?*

- * *What is the migration constructor and why do I need it?*
- * *What happens to the old copy of an array element after it migrates?*
- * *Is it possible to turn migratability on and off for an individual array element?*
- * *Is it possible to insist that a particular array element gets migrated at the next `AtSync()`?*
- * *When not using `AtSync` for LB, when does the LB start up? Where is the code that periodically checks if load balancing can be done?*
- * *Should I use `AtSync` explicitly, or leave it to the system?*
- *Charm++ Groups and Nodegroups*
 - * *What are groups and nodegroups used for?*
 - * *Should I use groups?*
 - * *Is it safe to use a local pointer to a group, such as from `ckLocalBranch`?*
 - * *What are migratable groups?*
 - * *Should I use nodegroups?*
 - * *What's the difference between groups and nodegroups?*
 - * *Do nodegroup entry methods execute on one fixed processor of the node, or on the next available processor?*
 - * *Are nodegroups single-threaded?*
 - * *Do we have to worry about two entry methods in an object executing simultaneously?*
- *Charm++ Messages*
 - * *What are messages?*
 - * *Should I use messages?*
 - * *What is the best way to pass pointers in a message?*
 - * *Can I allocate a message on the stack?*
 - * *Do I need to delete messages that are sent to me?*
 - * *Do I need to delete messages that I allocate and send?*
 - * *What can a variable-length message contain?*
 - * *Do I need to delete the arrays in variable-length messages?*
 - * *What are priorities?*
 - * *Can messages have multiple inheritance in Charm++?*
- *PUP Framework*
 - * *How does one write a pup for a dynamically allocated 2-dimensional array?*
 - * *When using automatic allocation via `PUP::able`, what do these calls mean?
`PUPable_def(parent); PUPable_def(child);`*
 - * *What is the difference between `p|data;` and `p(data);`? Which one should I use?*
- *Other PPL Tools, Libraries and Applications*
 - * *What is Structured Dagger?*

- * *What is Adaptive MPI?*
- * *What is Charisma?*
- * *Does Projections use wall time or CPU time?*
- *Debugging*
 - * *How can I debug Charm++ programs?*
 - * *How do I use charmdebug?*
 - * *Can I use distributed debuggers like Allinea DDT and RogueWave TotalView?*
 - * *How do I use gdb with Charm++ programs?*
 - * *When I try to use the ++debug option I get: remote host not responding... connection closed*
 - * *My debugging printouts seem to be out of order. How can I prevent this?*
 - * *Is there a way to flush the print buffers in Charm++ (like fflush())?*
 - * *My Charm++ program is causing a seg fault, and the debugger shows that it's crashing inside malloc or printf or fopen!*
 - * *Everything works fine on one processor, but when I run on multiple processors it crashes!*
 - * *I get the error: "Group ID is zero-- invalid!". What does this mean?*
 - * *I get the error: Null-Method Called. Program may have Unregistered Module!! What does this mean?*
 - * *When I run my program, it gives this error:*
 - * *When I run my program, sometimes I get a Hangup, and sometimes Bus Error. What do these messages indicate?*
- *Versions and Ports*
 - * *Has Charm++ been ported to use MPI underneath? What about OpenMP?*
 - * *How complicated is porting Charm++/Converse?*
 - * *If the source is available how feasible would it be for us to do ports ourselves?*
 - * *To what platform has Charm++/Converse been ported to?*
 - * *Is it hard to port Charm++ programs to different machines?*
 - * *How should I approach portability of C language code?*
 - * *How should I approach portability and performance of C++ language code?*
 - * *Why do I get a link error when mixing Fortran and C/C++?*
 - * *How does parameter passing work between Fortran and C?*
 - * *How do I use Charm++ on Xeon Phi?*
 - * *How do I use Charm++ on GPUs?*
- *Converse Programming*
 - * *What is Converse? Should I use it?*
 - * *How much does getting a random number generator "right" matter?*
 - * *What should I use to get a proper random number generator?*

– *Charm++ and Converse Internals*

- * *How is the Charm++ source code organized and built?*
- * *I just changed the Charm++ core. How do I recompile Charm++?*
- * *Do we have a #define charm_version somewhere? If not, which version number should I use for the current version?*

For answers to questions not on this list, please create an issue or discussion on our [GitHub](#).

6.1 Big Questions

6.1.1 What is Charm++?

Charm++ is a runtime library to let C++ objects communicate with each other efficiently. The programming model is thus an asynchronous message driven paradigm, like Java RMI, or RPC; but it is targeted towards tightly coupled, high-performance parallel machines. Unlike MPI's "single program, multiple data" (SPMD) programming and execution model, Charm++ programs do not proceed in lockstep. The flow of control is determined by the order in which remote method invocations occur. This can be controlled by the user through Structure Control Flow using *Structured Dagger*, or *Charisma*.

Charm++ has demonstrated scalability up to hundreds of thousands of processors, and provides extremely advanced load balancing and object migration facilities.

6.1.2 Can Charm++ parallelize my serial program automatically?

No.

Charm++ is used to write *explicitly parallel* programs—we don't have our own compiler, so we don't do automatic parallelization. We've found automatic parallelization useful only for a small range of very regular numerical applications.

However, you should *not* have to throw away your serial code; normally only a small fraction of a large program needs to be changed to enable parallel execution. In particular, Charm++'s support for object-oriented programming and high-level abstractions such as Charm++ Arrays make it simpler and more expressive than many other parallel languages. So you will have to write some new code, but not as much as you might think. This is particularly true when using one of the Charm++ [frameworks](#).

6.1.3 I can already write parallel applications in MPI. Why should I use Charm++?

Charm++ provides several extremely sophisticated features, such as application-independent object migration, fault tolerance, power awareness, and automatic overlap of communication with computation, that are very difficult to provide in MPI. If you have a working MPI code but have scalability problems because of dynamic behavior, load imbalance, or communication costs, Charm++ might dramatically improve your performance. You can even run your MPI code on Charm++ unchanged using [AMPI](#).

6.1.4 Will Charm++ run on my machine?

Yes.

Charm++ supports both shared-memory and distributed-memory machines, SMPs and non-SMPs. In particular, we support serial machines, Windows machines, Apple machines, ARM machines, clusters connected via Ethernet, or Infiniband, IBM Power series and BlueGene/Q, Cray XC/XE/XK series, and any machine that supports MPI. We normally do our development on Linux workstations, and our testing on large parallel machines. Programs written using Charm++ will run on any supported machine.

6.1.5 Does anybody actually use Charm++?

Several large applications use Charm++.

- The large, production-quality molecular dynamics application [NAMD](#).
- The cosmological simulator [ChaNGa](#).
- The atomistic simulation framework [OpenAtom](#).
- We have significant collaborations with groups in Materials Science, Chemistry, Astrophysics, Network Simulation, Operation Research, Contagion Effects, in Illinois, New York, California, Washington, and Virginia. See also [Applications](#) for a more complete list.

6.1.6 Who created Charm++?

Prof. [L.V. Kale](#), of the [Computer Science Department](#) of the [University of Illinois at Urbana-Champaign](#), and his research group, the [Parallel Programming Lab](#). Nearly a hundred people have contributed something to the project over the course of approximately 20 years; a partial list of contributors appears on the [people](#) page.

6.1.7 What is the future of Charm++?

Our research group of approximately twenty people are actively engaged in maintaining and extending Charm++; and in particular the Charm++ [frameworks](#). Several other groups are dependent on Charm++, so we expect to continue improving Charm++ indefinitely.

6.1.8 How is Charm++ Licensed?

Charm++ is open-source and free for research, educational, and academic use. The University of Illinois retains the copyright to the software, and requires a license for any commercial redistribution of our software. The actual, legal license is included with Charm++ (in `charm/LICENSE`). Contact [Charmworks, Inc.](#) for commercial support and licensing of Charm++ and AMPI.

6.1.9 I have a suggestion/feature request/bug report. Who should I send it to?

Please open an issue or discussion on our [GitHub](#). We are always glad to get feedback on our software.

6.2 Installation and Usage

6.2.1 How do I get Charm++?

See our [download](#) page.

6.2.2 Should I use the GIT version of Charm++?

The developers of Charm++ routinely use the latest GIT versions, and most of the time this is the best case. Occasionally something breaks, but the GIT version will likely contain bug fixes not found in the releases.

6.2.3 How do I compile Charm++?

Run the interactive build script `./build` with no extra arguments. If this fails, please open an issue on our [GitHub](#) with the problem. Include the build line used (this is saved automatically in `smart-build.log`)

If you have a very unusual machine configuration, you will have to run `./build -help` to list all possible build options. You will then choose the closest architecture, and then you may have to modify the associated `conf-mach.sh` and `conv-mach.h` files in `src/arch` to point to your desired compilers and options. If you develop a significantly different platform, please open a pull request on our [GitHub](#) with the modified files so we can include it in the distribution.

6.2.4 How do I compile AMPI?

Run the build script `./build` and choose the option for building “Charm++ and AMPI,” or just replace “charm++” in your full build command with “AMPI”, as in `./build AMPI netlrts-linux-x86_64`.

6.2.5 Can I remove part of charm tree after compilation to free disk space?

Yes. Keep `src`, `bin`, `lib`, `lib_so`, `include`, `tmp`. You will not need `tests`, `examples`, `doc`, `contrib` for normal usage once you have verified that your build is functional.

6.2.6 If the interactive script fails, how do I compile Charm++?

See the Charm manual for *Installing Charm++*.

6.2.7 How do I specify the processors I want to use?

On machines where MPI has already been wired into the job system, use the `-mpiexec` flag and `-np` arguments.

For the netlrts- versions, you need to write a nodelist file which lists all the machine hostnames available for parallel runs.

```
group main
  host foo1
  host foo2 ++cpus 4
  host foo3.bar.edu
```

For the MPI version, you need to set up an MPI configuration for available machines as for normal MPI applications.

You can specify the exact cores to use on each node using the `+pemap` option. When running in SMP or multicore mode, this applies to the worker threads only, not communication threads. To specify the placement of communication threads, use the `+commap` option. For example, to place 8 threads on 2 nodes (16 threads total) with the comm thread on core 0 and the worker threads on cores 1 - 7, you would use the following command:

```
./charmrun +p14 ./pgm ++ppn 7 +commap 0 +pemap 1-7
```

See *SMP Options* of the Charm++ manual for more information.

6.2.8 How do I use *ssh* instead of the deprecated *rsh*?

You need to set up your `.ssh/authorized_keys` file correctly. Setup no-password logins using *ssh* by putting the correct host key (*ssh-keygen*) in the file `.ssh/authorized_keys`.

Finally, in the `.nodelist` file, you specify the shell to use for remote execution of a program using the keyword `++shell`.

```
group main ++shell ssh
  host foo1
  host foo2
  host foo3
```

6.2.9 Can I use the serial library X from a Charm program?

Yes. Some of the known working serial libraries include:

- The Tcl/Tk interpreter (in NAMD)
- The Python interpreter (in Cosmo prototype)
- OpenGL graphics (in graphics demos)
- Metis mesh partitioning (included with charm)
- ATLAS, BLAS, LAPACK, ESSL, FFTW, MASSV, ACML, MKL, BOOST

In general, any serial library should work fine with Charm++.

6.2.10 How do I get the command-line switches available for a specific program?

Try

```
./charmrun ./pgm --help
```

to see a list of parameters at the command line. The `charmrun` arguments are documented in *Launching Programs with `charmrun`*. The arguments for the installed libraries are listed in the library manuals.

6.2.11 What should I do if my program hangs while gathering CPU topology information at startup?

This is an indication that your cluster's DNS server is not responding properly. Ideally, the DNS resolver configured to serve your cluster nodes should be able to rapidly map their hostnames to their IP addresses. As an immediate workaround, you can run your program with the `+skip_cpu_topology` flag, at the possible cost of reduced performance. Another workaround is installing and running `nsd`, the "name service caching daemon", on your cluster nodes; this may add some noise on your systems and hence reduce performance. A third workaround is adding the addresses and names of all cluster nodes in each node's `/etc/hosts` file; this poses maintainability problems for ongoing system administration.

6.3 Basic Charm++ Programming

6.3.1 What's the basic programming model for Charm++?

Parallel objects using “Asynchronous Remote Method Invocation”:

Asynchronous in that you *do not block* until the method returns-the caller continues immediately.

Remote in that the two objects may be separated by a network.

Method Invocation in that it's just C++ classes calling each other's methods.

6.3.2 What is an “entry method”?

Entry methods are all the methods of a chore where messages can be sent by other chares. They are declared in the .ci files, and they must be defined as public methods of the C++ object representing the chore.

6.3.3 When I invoke a remote method, do I block until that method returns?

No! This is one of the biggest differences between Charm++ and most other “remote procedure call” systems like, Java RMI, or RPC. “Invoke an asynchronous method” and “send a message” have exactly the same semantics and implementation. Since the invoking method does not wait for the remote method to terminate, it normally cannot receive any return value. (see later for a way to return values)

6.3.4 Why does Charm++ use asynchronous methods?

Asynchronous method invocation is more efficient because it can be implemented as a single message send. Unlike with synchronous methods, thread blocking and unblocking and a return message are not needed.

Another big advantage of asynchronous methods is that it's easy to make things run in parallel. If I execute:

```
a->foo();
b->bar();
```

Now foo and bar can run at the same time; there's no reason bar has to wait for foo.

6.3.5 Can I make a method synchronous? Can I then return a value?

Yes. If you want synchronous methods, so the caller will block, use the `[sync]` keyword before the method in the .ci file. This requires the sender to be a threaded entry method, as it will be suspended until the callee finishes. Sync entry methods are allowed to return values to the caller.

6.3.6 What is a threaded entry method? How does one make an entry method threaded?

A threaded entry method is an entry method for a chore that executes in a separate user-level thread. It is useful when the entry method wants to suspend itself (for example, to wait for more data). Note that threaded entry methods have nothing to do with kernel-level threads or pthreads; they run in user-level threads that are scheduled by Charm++ itself.

In order to make an entry method threaded, one should add the keyword *threaded* withing square brackets after the *entry* keyword in the interface file:

```
module M {  
  chare X {  
    entry [threaded] E1(void);  
  };  
};
```

6.3.7 If I don't want to use threads, how can an asynchronous method return a value?

The usual way to get data back to your caller is via another invocation in the opposite direction:

```
void A::start(void) {  
  b->giveMeSomeData();  
}  
void B::giveMeSomeData(void) {  
  a->hereIsTheData(data);  
}  
void A::hereIsTheData(myclass_t data) {  
  ...use data somehow...  
}
```

This is contorted, but it exactly matches what the machine has to do. The difficulty of accessing remote data encourages programmers to use local data, bundle outgoing requests, and develop higher-level abstractions, which leads to good performance and good code.

6.3.8 Isn't there a better way to send data back to whoever called me?

The above example is very non-modular, because *b* has to know that *a* called it, and what method to call a back on. For this kind of request/response code, you can abstract away the “where to return the data” with a *CkCallback* object:

```
void A::start(void) {  
  b->giveMeSomeData(CkCallback(CkIndex_A::hereIsTheData,thisProxy));  
}  
void B::giveMeSomeData(CkCallback returnDataHere) {  
  returnDataHere.send(data);  
}  
void A::hereIsTheData(myclass_t data) {  
  ...use data somehow...  
}
```

Now *b* can be called from several different places in *a*, or from several different modules.

6.3.9 Why should I prefer the callback way to return data rather than using [sync] entry methods?

There are a few reasons for that:

- The caller needs to be threaded, which implies some overhead in creating the thread. Moreover the threaded entry method will suspend waiting for the data, preventing any code after the remote method invocation to proceed in parallel.
- Threaded entry methods are still methods of an object. While they are suspended other entry methods for the same object (or even the same threaded entry method) can be called. This allows for potential problems if the suspending method does leave some objects in an inconsistent state.

- Finally, and probably most important, `[sync]` entry methods can only be used to return a value that can be computed by a single chare. When more flexibility is needed, such in cases where the resulting value needs to the contribution of multiple objects, the callback methodology is the only one available. The caller could for example send a broadcast to a chare array, which will use a reduction to collect back the results after they have been computed.

6.3.10 How does the initialization in Charm work?

Each processor executes the following operations strictly in order:

1. All methods registered as *initnode*;
2. All methods registered as *initproc*;
3. On processor zero, all *mainchares* constructor method is invoked (the ones taking a `CkArgMsg*`);
4. The read-onlies are propagated from processor zero to all other processors;
5. The nodegroups are created;
6. The groups are created. During this phase, for all the chare arrays have been created with a block allocation, the corresponding array elements are instantiated;
7. Initialization terminated and all messages are available for processing, including the messages responsible for the instantiation of array elements manually inserted.

This implies that you can assume that the previous steps has completely finished before the next one starts, and any side effect from all the previous steps are committed (and can therefore be used).

Inside a single step there is no order guarantee. This implies that, for example, two groups allocated from *mainchare* can be instantiated in any order. The only exception to this is processor zero, where chare objects are instantiated immediately when allocated in the *mainchare*, i.e if two groups are allocated, their order is fixed by the allocation order in the *mainchare* constructing them. Again, this is only valid for processor zero, and in no other processor this assumption should be made.

To notice that if array elements are allocated in block (by specifying the number of elements at the end of the `ckNew` function), they are all instantiated before normal execution is resumed; if manual insertion is used, each element can be constructed at any time on its home processor, and not necessarily before other regular communication messages have been delivered to other chares (including other array elements part of the same array).

6.3.11 Does Charm++ support C and Fortran?

C and Fortran routines can be called from Charm++ using the usual API conventions for accessing them from C++. AMPI supports Fortran directly, but direct use of Charm++ semantics from Fortran is at an immature stage, contact us if you are interested in pursuing this further.

6.3.12 What is a proxy?

A proxy is a local C++ class that represents a remote C++ class. When you invoke a method on a proxy, it sends the request across the network to the real object it represents. In Charm++, all communication is done using proxies.

A proxy class for each of your classes is generated based on the methods you list in the `.ci` file.

6.3.13 What are the different ways one can create proxies?

Proxies can be:

- Created using `ckNew`. This is the only method that actually creates a new parallel object. “`CProxy_A::ckNew(...)`” returns a proxy.
- Copied from an existing proxy. This happens when you assign two proxies or send a proxy in a message.
- Created from a “handle”. This happens when you say “`CProxy_A p=thishandle;`”
- Created uninitialized. This is the default when you say “`CProxy_A p;`”. You’ll get a runtime error “proxy has not been initialized” if you try to use an uninitialized proxy.

6.3.14 What is wrong if I do `A *ap = new CProxy_A(handle)`?

This will not compile, because a `CProxy_A` is not an `A`. What you want is `CProxy_A *ap = new CProxy_A(handle)`.

6.3.15 Why is the `def.h` usually included at the end? Is it necessary or can I just include it at the beginning?

You can include the `def.h` file once you’ve actually declared everything it will reference- all your chares and readonly variables. If your chares and readonlies are in your own header files, it is legal to include the `def.h` right away.

However, if the class declaration for a chare isn’t visible when you include the `def.h` file, you’ll get a confusing compiler error. This is why we recommend including the `def.h` file at the end.

6.3.16 How can I use a global variable across different processors?

Make the global variable “readonly” by declaring it in the `.ci` file. Remember also that read-onlies can be safely set only in the mainchare constructor. Any change after the mainchare constructor has finished will be local to the processor that made the change. To change a global variable later in the program, every processor must modify it accordingly (e.g by using a chare group. Note that chare arrays are not guaranteed to cover all processors)

6.3.17 Can I have a class static read-only variable?

One can have class-static variables as read-onlies. Inside a chare, group or array declaration in the `.ci` file, one can have a readonly variable declaration. Thus:

```
chare someChare {  
  ...  
  readonly CkGroupID someGroup;  
  ...  
};
```

is fine. In the `.h` declaration for `class someChare`, you will have to put `someGroup` as a public static variable, and you are done.

You then refer to the variable in your program as `someChare::someGroup`.

6.3.18 How do I measure the time taken by a program or operation?

You can use `CkWallTimer()` to determine the time on some particular processor. To time some parallel computation, you need to call `CkWallTimer` on some processor, do the parallel computation, then call `CkWallTimer` again on the same processor and subtract.

6.3.19 What do `CmiAssert` and `CkAssert` do?

These are just like the standard C++ `assert` calls in `<assert.h>`- they call abort if the condition passed to them is false.

We use our own version rather than the standard version because we have to call `CkAbort`, and because we can turn our asserts off when `-with-production` is used on the build line. These assertions are specifically controlled by `-enable-error-checking` or `-disable-error-checking`. The `-with-production` flag implies `-disable-error-checking`, but it can still be explicitly enabled with `-enable-error-checking`.

6.3.20 Can I know how many messages are being sent to a chore?

No.

There is no nice library to solve this problem, as some messages might be queued on the receiving processor, some on the sender, and some on the network. You can still:

- Send a return receipt message to the sender, and wait until all the receipts for the messages sent have arrived, then go to a barrier;
- Do all the sends, then wait for quiescence.

6.3.21 What is “quiescence”? How does it work?

Quiescence is When nothing is happening anywhere on the parallel machine.

A low-level background task counts sent and received messages. When, across the machine, all the messages that have been sent have been received, and nothing is being processed, quiescence is triggered.

6.3.22 Should I use quiescence detection?

Probably not.

See the [Completion Detection](#) section of the manual for instructions on a more local inactivity detection scheme.

In some ways, quiescence is a very strong property (it guarantees *nothing* is happening *anywhere*) so if some other library is doing something, you won't reach quiescence. In other ways, quiescence is a very weak property, since it doesn't guarantee anything about the state of your application like a reduction does, only that nothing is happening. Because quiescence detection is on the one hand so strong it breaks modularity, and on the other hand is too weak to guarantee anything useful, it's often better to use something else.

Often global properties can be replaced by much easier-to-compute local properties. For example, my object could wait until all *its* neighbors have sent it messages (a local property my object can easily detect by counting message arrivals), rather than waiting until *all* neighbor messages across the whole machine have been sent (a global property that's difficult to determine). Sometimes a simple reduction is needed instead of quiescence, which has the benefits of being activated explicitly (each element of a chore array or chore group has to call `contribute`) and allows some data to be collected at the same time. A reduction is also a few times faster than quiescence detection. Finally, there are a few situations, such as some tree-search problems, where quiescence detection is actually the most sensible, efficient solution.

6.4 Charm++ Arrays

6.4.1 How do I know which processor a chare array element is running on?

At any given instant, you can call `CkMyPe()` to find out where you are. There is no reliable way to tell where another array element is; even if you could find out at some instant, the element might immediately migrate somewhere else!

6.4.2 Should I use Charm++ Arrays in my program?

Yes! Most of your computation should happen inside array elements. Arrays are the main way to automatically balance the load using one of the load balancers available.

6.4.3 Is there a property similar to `thisIndex` containing the chare array's dimensions or total number of elements?

No. In more sophisticated Charm++ algorithms and programs, array dimensions are a dynamic property, and since Charm++ operates in a distributed system context, any such value would potentially be stale between access and use.

If the array in question has a fixed size, then that size can be passed to its elements as an argument to their constructor or some later entry method call. Otherwise, the object(s) creating the chare array elements should perform a reduction to count them.

6.4.4 How many array elements should I have per processor?

To do load balancing, you need more than one array element per processor. To keep the time and space overheads reasonable, you probably don't want more than a few thousand array elements per processor. The optimal value depends on the program, but is usually between 10 and 100. If you come from an MPI background, this may seem like a lot.

6.4.5 What does the term reduction refer to?

You can *reduce* a set of data to a single value. For example, finding the sum of values, where each array element contributes a value to the final sum. Reductions are supported directly by Charm++ arrays, and some operations most commonly used are predefined. Other more complicated reductions can implement if needed.

6.4.6 Can I do multiple reductions on an array?

You *can* have several reductions happen one after another; but you *cannot* mix up the execution of two reductions over the same array. That is, if you want to reduce A, then B, every array element has to contribute to A, then contribute to B; you cannot have some elements contribute to B, then contribute to A.

In addition, *Tuple* reductions provide a way of performing multiple different reductions using the same reduction message. See the *Built-in Reduction Types* section of the manual for more information on Tuple reductions.

6.4.7 Does Charm++ do automatic load balancing without the user asking for it?

No. You only get load balancing if you explicitly ask for it by linking in one or more load balancers with *-balancer* link-time option. If you link in more than one load balancer, you can select from the available load balancers at runtime

with the *+balancer* option. In addition, you can use Metabaler with the *+MetaLB* option to automatically decide when to invoke the load balancer, as described in [Available Load Balancing Strategies](#).

6.4.8 What is the migration constructor and why do I need it?

The migration constructor (a constructor that takes `CkMigrateMessage *` as parameter) is invoked when an array element migrates to a new processor, or when chares or group instances are restored from a checkpoint. If there is anything you want to do when you migrate, you could put it here.

A migration constructor need not be defined for any given chare type. If you try to migrate instances of a chare type that lacks a migration constructor, the runtime system will abort the program with an error message.

The migration constructor should not be declared in the *.ci* file. Of course the array element will require also at least one regular constructor so that it can be created, and these must be declared in the *.ci* file.

6.4.9 What happens to the old copy of an array element after it migrates?

After sizing and packing a migrating array element, the array manager deletes the old copy. As long as all the array element destructors in the non-leaf nodes of your inheritance hierarchy are *virtual destructors*, with declaration syntax:

```
class foo : ... {
    ...
    virtual ~foo(); // <- virtual destructor
};
```

then everything will get deleted properly.

Note that deleting things in a packing pup happens to work for the current array manager, but *WILL NOT* work for checkpointing, debugging, or any of the (many) other uses for packing puppers we might dream up - so DON'T DO IT!

6.4.10 Is it possible to turn migratability on and off for an individual array element?

Yes, call *setMigratable(false)*; in the constructor.

6.4.11 Is it possible to insist that a particular array element gets migrated at the next *AtSync()*?

No, but a manual migration can be triggered using *migrateMe*.

6.4.12 When not using *AtSync* for LB, when does the LB start up? Where is the code that periodically checks if load balancing can be done?

If not using *usesAtSync*, load balancing will not run by default. If one does not want to use *AtSync*, but instead run in a mode where load balancing is automatically run periodically, the user must run with the *+LBPeriod <time in seconds>* runtime option. Load balancing will be invoked automatically by the runtime system, waiting at least the specified value of time between successive calls. In this load balancing mode, users have to make sure all migratable objects are always ready to migrate (e.g. not depending on a global variable which cannot be migrated).

6.4.13 Should I use AtSync explicitly, or leave it to the system?

You almost certainly want to use AtSync directly. In most cases there are points in the execution where the memory in use by a chore is bigger due to transitory data, which does not need to be transferred if the migration happens at predefined points.

6.5 Charm++ Groups and Nodegroups

6.5.1 What are groups and nodegroups used for?

They are used for optimizations at the processor and node level respectively.

6.5.2 Should I use groups?

Probably not. People with an MPI background often overuse groups, which results in MPI-like Charm++ programs. Arrays should generally be used instead, because arrays can be migrated to achieve load balance.

Groups tend to be most useful in constructing communication optimization libraries. For example, all the array elements on a processor can contribute something to their local group, which can then send a combined message to another processor. This can be much more efficient than having each array element send a separate message.

6.5.3 Is it safe to use a local pointer to a group, such as from ckLocalBranch?

Yes. Groups never migrate, so a local pointer is safe. The only caveat is to make sure *you* don't migrate without updating the pointer.

A local pointer can be used for very efficient access to data held by a group.

6.5.4 What are migratable groups?

Migratable groups are declared so by adding the “[migratable]” attribute in the .ci file. They *cannot* migrate from one processor to another during normal execution, but only to disk for checkpointing purposes.

Migratable groups must declare a migration constructor (taking `CkMigrateMessage *` as a parameter) and a pup routine. The migration constructor *must* call the superclass migration constructor as in this example:

```
class MyGroup : public CBase_MyGroup {
...
    MyGroup (CkMigrateMessage *msg) : CBase_MyGroup(msg) { }
...
}
```

6.5.5 Should I use nodegroups?

Almost certainly not. You should use arrays for most computation, and even quite low-level communication optimizations are often best handled by groups. Nodegroups are very difficult to get right.

6.5.6 What's the difference between groups and nodegroups?

There's one group element per processor (CkNumPes() elements); and one nodegroup element per node (CkNumNodes() elements). Because they execute on a node, nodegroups have very different semantics from the rest of Charm++.

Note that on a non-SMP machine, groups and nodegroups are identical.

6.5.7 Do nodegroup entry methods execute on one fixed processor of the node, or on the next available processor?

Entries in node groups execute on the next available processor. Thus, if two messages were sent to a branch of a nodegroup, two processors could execute one each simultaneously.

6.5.8 Are nodegroups single-threaded?

No. They *can* be accessed by multiple threads at once.

6.5.9 Do we have to worry about two entry methods in an object executing simultaneously?

Yes, which makes nodegroups different from everything else in Charm++.

If a nodegroup method accesses a data structure in a non-threadsafe way (such as writing to it), you need to lock it, for example using a CmiNodeLock.

6.6 Charm++ Messages

6.6.1 What are messages?

A bundle of data sent, via a proxy, to another chare. A message is a special kind of heap-allocated C++ object.

6.6.2 Should I use messages?

It depends on the application. We've found parameter marshalling to be less confusing and error-prone than messages for small parameters. Nevertheless, messages can be more efficient, especially if you need to buffer incoming data, or send complicated data structures (like a portion of a tree).

6.6.3 What is the best way to pass pointers in a message?

You can't pass pointers across processors. This is a basic fact of life on distributed-memory machines.

You can, of course, pass a copy of an object referenced via a pointer across processors-either dereference the pointer before sending, or use a varsize message.

6.6.4 Can I allocate a message on the stack?

No. You must allocate messages with *new*.

6.6.5 Do I need to delete messages that are sent to me?

Yes, or you will leak memory! If you receive a message, you are responsible for deleting it. This is exactly opposite of parameter marshalling, and much common practice. The only exception are entry methods declared as `[nokeep]`; for these the system will free the message automatically at the end of the method.

6.6.6 Do I need to delete messages that I allocate and send?

No, this will certainly corrupt both the message and the heap! Once you've sent a message, it's not yours any more. This is again exactly the opposite of parameter marshalling.

6.6.7 What can a variable-length message contain?

Variable-length messages can contain arrays of any type, both primitive type or any user-defined type. The only restriction is that they have to be 1D arrays.

6.6.8 Do I need to delete the arrays in variable-length messages?

No, this will certainly corrupt the heap! These arrays are allocated in a single contiguous buffer together with the message itself, and is deleted when the message is deleted.

6.6.9 What are priorities?

Priorities are special values that can be associated with messages, so that the Charm++ scheduler will generally prefer higher priority messages when choosing a buffered message from the queue to invoke as an entry method. Priorities are often respected by Charm++ scheduler, but for correctness, a program must never rely upon any particular ordering of message deliveries. Messages with priorities are typically used to encourage high performance behavior of an application.

For integer priorities, the smaller the priority value, the higher the priority of the message. Negative value are therefore higher priority than positive ones. To enable and set a message's priority there is a special *new* syntax and *CkPriorityPtr* function; see the manual for details. If no priority is set, messages have a default priority of zero.

6.6.10 Can messages have multiple inheritance in Charm++?

Yes, but you probably shouldn't. Perhaps you want to consider using *Generic and Meta Programming with Templates* techniques with templated chares, methods, and/or messages instead.

6.7 PUP Framework

6.7.1 How does one write a pup for a dynamically allocated 2-dimensional array?

The usual way: pup the size(s), allocate the array if unpacking, and then pup all the elements.

For example, if you have a 2D grid like this:

```

class foo {
private:
    int wid,ht;
    double **grid;
    ...other data members

    //Utility allocation/deallocation routines
    void allocateGrid(void) {
        grid=new double*[ht];
        for (int y=0;y<ht;y++)
            grid[y]=new double[wid];
    }
    void freeGrid(void) {
        for (int y=0;y<ht;y++)
            delete[] grid[y];
        delete[] grid;
        grid=NULL;
    }

public:
    //Regular constructor
    foo() {
        ...set wid, ht...
        allocateGrid();
    }
    //Migration constructor
    foo(CkMigrateMessage *) {}
    //Destructor
    ~foo() {
        freeGrid();
    }

    //pup method
    virtual void pup(PUP::er &p) {
        p(wid); p(ht);
        if (p.isUnpacking()) {
            //Now that we know wid and ht, allocate grid
            allocateGrid(wid,ht);
        }
        //Pup grid values element-by-element
        for (int y=0;y<ht;y++)
            PUParray(p, grid[y], wid);
        ...pup other data members...
    }
};

```

6.7.2 When using automatic allocation via PUP::able, what do these calls mean?

PUPable_def(parent); PUPable_def(child);

For the automatic allocation described in *Automatic allocation via “PUP::able”* of the manual, each class needs four things:

- A migration constructor
- PUPable_decl(className) in the class declaration in the .h file
- PUPable_def(className) at file scope in the .C file

- `PUPable_reg(className)` called exactly once on every node. You typically use the *initproc* mechanism to call these.

See `charm/tests/charm++/megatest/marshall.[hC]` for an executable example.

6.7.3 What is the difference between `p|data;` and `p(data);`? Which one should I use?

For most system- and user-defined structure *someHandle*, you want `p|someHandle;` instead of `p(someHandle);`

The reason for the two incompatible syntax varieties is that the bar operator can be overloaded *outside* `pup.h` (just like the `std::ostream`'s `operator<<`); while the parenthesis operator can take multiple arguments (which is needed for efficiently PUPing arrays).

The bar syntax will be able to copy *any* structure, whether it has a `pup` method or not. If there is no `pup` method, the C++ operator overloading rules decay the bar operator into packing the *bytes* of the structure, which will work fine for simple types on homogeneous machines. For dynamically allocated structures or heterogeneous migration, you'll need to define a `pup` method for all packed classes/structures. As an added benefit, the same `pup` methods will get called during parameter marshalling.

6.8 Other PPL Tools, Libraries and Applications

6.8.1 What is Structured Dagger?

Structured Dagger is a structured notation for specifying intra-process control dependencies in message-driven programs. It combines the efficiency of message-driven execution with the explicitness of control specification. Structured Dagger allows easy expression of dependencies among messages and computations and also among computations within the same object using `when-blocks` and various structured constructs. See the Charm++ manual for the details.

6.8.2 What is Adaptive MPI?

Adaptive MPI (AMPI) is an implementation of the MPI standard on top of Charm++. This allows MPI users to recompile their existing MPI applications with AMPI's compiler wrappers to take advantage of Charm++'s high level features, such as overdecomposition, overlap of communication and computation, dynamic load balancing, and fault tolerance. See the AMPI manual for more details on how AMPI works and how to use it.

6.8.3 What is Charisma?

Charisma++ is a prototype language for describing global view of control in a parallel program. It is designed to solve the problem of obscured control flow in the object-based model with Charm++.

6.8.4 Does Projections use wall time or CPU time?

Wall time.

6.9 Debugging

6.9.1 How can I debug Charm++ programs?

There are many ways to debug programs written in Charm++:

print By using `CkPrintf`, values from critical point in the program can be printed.

gdb This can be used both on a single processor, and in parallel simulations. In the latter, each processor has a terminal window with a `gdb` connected.

charmdebug This is the most sophisticated method to debug parallel programs in Charm++. It is tailored to Charm++ and it can display and inspect chare objects as well as messages in the system. Single *gdb*s can be attached to specific processors on demand.

6.9.2 How do I use charmdebug?

Currently `charmdebug` is tested to work only under `netlrts`- non-SMP versions. With other versions, testing is pending. To get the Charm Debug tool, check out the source code from the repository. This will create a directory named `ccs_tools`. Move to this directory and build Charm Debug.

```
$ git clone git@github.com:UIUC-PPL/ccs_tools
$ cd ccs_tools
$ ant
```

This will create the executable `bin/charmdebug`. To start, simply substitute “`charmdebug`” to “`charmrun`”:

```
$ ./charmdebug ./myprogram
```

You can find more detailed information in the debugger manual in *Charm++ Debugger*.

6.9.3 Can I use distributed debuggers like Alinea DDT and RogueWave TotalView?

Yes, on `mpi`- versions of Charm++. In this case, the program is a regular MPI application, and as such any tool available for MPI programs can be used. Notice that some of the internal data structures (like messages in queue) might be difficult to find.

Depending on your debugging needs, see the other notes about alternatives such as `CharmDebug` and directly-attached `gdb`.

6.9.4 How do I use gdb with Charm++ programs?

It depends on the machine. On the `netlrts`- versions of Charm++, like `netlrts-linux-x86_64`, you can just run the serial debugger:

```
$ gdb myprogram
```

If the problem only shows up in parallel, and you’re running on an X terminal, you can use the `++debug` or `++debug-no-pause` options of `charmrun` to get a separate window for each process:

```
$ export DISPLAY="myterminal:0"
$ ./charmrun ./myprogram +p2 ++debug
```

6.9.5 When I try to use the `++debug` option I get: `remote host not responding. .. connection closed`

First, make sure the program at least starts to run properly without `++debug` (i.e. `charmrun` is working and there are no problems with the program startup phase). You need to make sure that `gdb` or `dbx`, and `xterm` are installed on all the machines you are using (not the one that is running `charmrun`). If you are working from a Windows machine, you need an X-win application such as `exceed`. You need to set this up to give the right permissions for X windows. You need to make sure the `DISPLAY` environment variable on the remote machine is set correctly to your local machine. I recommend `ssh` and `putty`, because it will take care of the `DISPLAY` environment automatically, and you can set up `ssh` to use tunnels so that it even works from a private subnet (e.g. 192.168.0.8). Since the `xterm` is displayed from the node machines, you have to make sure they have the correct `DISPLAY` set. Again, setting up `ssh` in the `nodelist` file to spawn node programs should take care of that.

6.9.6 My debugging printouts seem to be out of order. How can I prevent this?

Printouts from different processors do not normally stay ordered. Consider the code:

```
...somewhere... {
    CkPrintf("cause\n");
    proxy.effect();
}
void effect(void) {
    CkPrintf("effect\n");
}
```

Though you might expect this code to always print “cause, effect”, you may get “effect, cause”. This can only happen when the cause and effect execute on different processors, so cause’s output is delayed.

If you pass the extra command-line parameter `+syncprint`, then `CkPrintf` actually blocks until the output is queued, so your printouts should at least happen in causal order. Note that this does dramatically slow down output.

6.9.7 Is there a way to flush the print buffers in Charm++ (like `fflush()`)?

Charm++ automatically flushes the print buffers every newline and at program exit. There is no way to manually flush the buffers at another point.

6.9.8 My Charm++ program is causing a seg fault, and the debugger shows that it’s crashing inside *malloc* or *printf* or *fopen*!

This isn’t a bug in the C library, it’s a bug in your program - you’re corrupting the heap. Link your program again with `-memory paranoid` and run it again in the debugger. `-memory paranoid` will check the heap and detect buffer over- and under-run errors, double-deletes, delete-garbage, and other common mistakes that trash the heap.

6.9.9 Everything works fine on one processor, but when I run on multiple processors it crashes!

It’s very convenient to do your testing on one processor (i.e., with `+p1`); but there are several things that only happen on multiple processors.

A single processor has just one set of global variables, but multiple processors have different global variables. This means on one processor, you can set a global variable and it stays set “everywhere” (i.e., right here!), while on two

processors the global variable never gets initialized on the other processor. If you must use globals, either set them on every processor or make them into *readonly* globals.

A single processor has just one address space, so you actually *can* pass pointers around between chares. When running on multiple processors, the pointers dangle. This can cause incredibly weird behavior - reading from uninitialized data, corrupting the heap, etc. The solution is to never, ever send pointers in messages - you need to send the data the pointer points to, not the pointer.

6.9.10 I get the error: “Group ID is zero-- invalid!”. What does this mean?

The *group* it is referring to is the chare group. This error is often due to using an uninitialized proxy or handle; but it’s possible this indicates severe corruption. Run with `++debug` and check if you just sent a message via an uninitialized proxy.

6.9.11 I get the error: Null-Method Called. Program may have Unregistered Module!! What does this mean?

You are trying to use code from a module that has not been properly initialized.

So, in the *.ci* file for your *mainmodule*, you should add an “extern module” declaration:

```
mainmodule whatever {
  extern module someModule;
  ...
}
```

6.9.12 When I run my program, it gives this error:

```
Charmrun: error on request socket-{}-
Socket closed before recv.
```

This means that the node program died without informing `charmrun` about it, which typically means a segmentation fault while in the interrupt handler or other critical communications code. This indicates severe corruption in Charm++’s data structures, which is likely the result of a heap corruption bug in your program. Re-linking with *-memory paranoid* may clarify the true problem.

6.9.13 When I run my program, sometimes I get a Hangup, and sometimes Bus Error. What do these messages indicate?

`Bus Error` and `Hangup` both are indications that your program is terminating abnormally, i.e. with an uncaught signal (SEGV or SIGBUS). You should definitely run the program with `gdb`, or use `++debug`. `Bus Errors` often mean there is an alignment problem, check if your compiler or environment offers support for detection of these.

6.10 Versions and Ports

6.10.1 Has Charm++ been ported to use MPI underneath? What about OpenMP?

Charm++ supports MPI and can use it as the underlying communication library. We have tested on MPICH, OpenMPI, and also most vendor MPI variants. Charm++ also has explicit support for SMP nodes in MPI version. Charm++ hasn’t

been ported to use OpenMP, but OpenMP can be used from Charm++.

6.10.2 How complicated is porting Charm++/Converse?

Depends. Hopefully, the porting only involves fixing compiler compatibility issues. The LRTS abstraction layer was designed to simplify this process and has been used for the MPI, Verbs, uGNI, PAMI and OFI layers. User level threads and Isomalloc support may require special platform specific support. Otherwise Charm++ is generally platform independent.

6.10.3 If the source is available how feasible would it be for us to do ports ourselves?

The source is always available, and you're welcome to make it run anywhere. Any kind of UNIX, Windows, and MacOS machine should be straightforward: just a few modifications to `charm/src/arch/.../conv-mach.h` (for compiler issues) and possibly a new *machine.c* (if there's a new communication system involved). However, porting to embedded hardware with a proprietary OS may be fairly difficult.

6.10.4 To what platform has Charm++/Converse been ported to?

Charm++/Converse has been ported to most UNIX and Linux OS, Windows, and MacOS.

6.10.5 Is it hard to port Charm++ programs to different machines?

Charm++ itself is fully portable, and should provide exactly the same interfaces everywhere (even if the implementations are sometimes different). Still, it's often harder than we'd like to port user code to new machines.

Many parallel machines have old or weird compilers, and sometimes a strange operating system or unique set of libraries. Hence porting code to a parallel machine can be surprisingly difficult.

Unless you're absolutely sure you will only run your code on a single, known machine, we recommend you be very conservative in your use of the language and libraries. "But it works with my gcc!" is often true, but not very useful.

Things that seem to work well everywhere include:

- Small, straightforward Makefiles. gmake-specific (e.g., "ifeq", filter variables) or convoluted makefiles can lead to porting problems and confusion. Calling `charm` instead of the platform-specific compiler will save you many headaches, as `charm` abstracts away the platform specific flags.
- Basically all of ANSI C and fortran 77 work everywhere. These seem to be old enough to now have the bugs largely worked out.
- C++ classes, inheritance, virtual methods, and namespaces work without problems everywhere. Not so uniformly supported are C++ templates, the STL, new-style C++ system headers, and the other features listed in the C++ question below.

6.10.6 How should I approach portability of C language code?

Our suggestions for Charm++ developers are:

- Avoid the nonstandard type "long long", even though many compilers happen to support it. Use `CMK_INT8` or `CMK_UINT8`, from `conv-config.h`, which are macros for the right thing. "long long" is not supported on many 64-bit machines (where "long" is 64 bits) or on Windows machines (where it's "`__int64`").

- The “long double” type isn’t present on all compilers. You can protect long double code with `#ifdef CMK_LONG_DOUBLE_DEFINED` if it’s really needed.
- Never use C++ “//” comments in C code, or headers included by C. This will not compile under many compilers.
- “bzero” and “bcopy” are BSD-specific calls. Use `memset` and `memcpy` for portable programs.

If you’re writing code that is expected to compile and run on Microsoft Windows using the Visual C++ compiler (e.g. modification to NAMD that you intend to submit for integration), that compiler has limited support for the C99 standard, and Microsoft recommends using C++ instead.

Many widely-used C compilers on HPC systems have limited support for the C11 standard. If you want to use features of C11 in your code, particularly `_Atomic`, we recommend writing the code in C++ instead, since C++11 standard support is much more ubiquitous.

6.10.7 How should I approach portability and performance of C++ language code?

The Charm++ system developers are conservative about which C++ standard version is relied upon in runtime system code and what features get used to ensure maximum portability across the broad range of HPC systems and the compilers used on them. Through version 6.8.x, the system code requires only limited support for C++11 features, specifically variadic templates and R-value references. From version 6.9 onwards, the system will require a compiler and standard library with at least full C++11 support.

A good reference for which compiler versions provide what level of standard support can be found at https://en.cppreference.com/w/cpp/compiler_support

Developers of several Charm++ applications have reported good results using features in more recent C++ standards, with the caveat of requiring that those applications be built with a sufficiently up-to-date C++ compiler.

The containers specified in the C++ standard library are generally designed to provide a very broad API that can be used correctly over highly-varied use cases. This often entails tradeoffs against the performance attainable for narrower use cases that some applications may have. The most visible of these concerns are the tension between strict iterator invalidation semantics and cache-friendly memory layout. We recommend that developers whose code includes container access in performance-critical elements explore alternative implementations, such as those published by EA, Google, and Facebook, or potentially write custom implementations tailored to their application’s needs.

In benchmarks across a range of compilers, we have found that avoiding use of exceptions (i.e. `throw/catch`) and disabling support for them with compiler flags can produce higher-performance code, especially with aggressive optimization settings enabled. The runtime system does not use exceptions internally. If your goal as an application developer is to most efficiently use large-scale computational resources, we recommend alternative error-handling strategies.

6.10.8 Why do I get a link error when mixing Fortran and C/C++?

Fortran compilers “mangle” their routine names in a variety of ways. `g77` and most compilers make names all lowercase, and append an underscore, like “foo_”. The IBM `xlf` compiler makes names all lowercase without an underscore, like “foo”. Absoft `f90` makes names all uppercase, like “FOO”.

If the Fortran compiler expects a routine to be named “foo_”, but you only define a C routine named “foo”, you’ll get a link error (“undefined symbol foo_”). Sometimes the UNIX command-line tool `nm` (list symbols in a `.o` or `.a` file) can help you see exactly what the Fortran compiler is asking for, compared to what you’re providing.

Charm++ automatically detects the Fortran name mangling scheme at configure time, and provides a C/C++ macro “FTN_NAME”, in “charm-api.h”, that expands to a properly mangled Fortran routine name. You pass the FTN_NAME macro two copies of the routine name: once in all uppercase, and again in all lowercase. The FTN_NAME macro then picks the appropriate name and applies any needed underscores. “charm-api.h” also includes a macro “FLINKAGE”

that makes the symbol linkable from fortran (in C++, this expands to extern “C”), so a complete Fortran subroutine looks like in C or C++:

```
FLINKAGE void FTN_NAME(FOO,foo) (void);
```

This same syntax can be used for C/C++ routines called from fortran, or for calling fortran routines from C/C++. We strongly recommend using FTN_NAME instead of hardcoding your favorite compiler’s name mangling into the C routines.

If designing an API with the same routine names in C and Fortran, be sure to include both upper and lowercase letters in your routine names. This way, the C name (with mixed case) will be different from all possible Fortran manglings (which all have uniform case). For example, a routine named “foo” will have the same name in C and Fortran when using the IBM xlf compilers, which is bad because the C and Fortran versions should take different parameters. A routine named “Foo” does not suffer from this problem, because the C version is “Foo, while the Fortran version is “foo_”, “foo”, or “FOO”.

6.10.9 How does parameter passing work between Fortran and C?

Fortran and C have rather different parameter-passing conventions, but it is possible to pass simple objects back and forth between Fortran and C:

- Fortran and C/C++ data types are generally completely interchangeable:

C/C++ Type	Fortran Type
int	INTEGER, LOGICAL
double	DOUBLE PRECISION, REAL*8
float	REAL, REAL*4
char	CHARACTER

- Fortran internally passes everything, including constants, integers, and doubles, by passing a pointer to the object. Hence a fortran “INTEGER” argument becomes an “int *” in C/C++:

```
/* Fortran */
SUBROUTINE BAR(i)
  INTEGER :: i
  x=i
END SUBROUTINE
```

```
/* C/C++ */
FLINKAGE void FTN_NAME(BAR,bar) (int *i) {
  x=*i;
}
```

- 1D arrays are passed exactly the same in Fortran and C/C++: both languages pass the array by passing the address of the first element of the array. Hence a fortran “INTEGER, DIMENSION(:)” array is an “int *” in C or C++. However, Fortran programmers normally think of their array indices as starting from index 1, while in C/C++ arrays always start from index 0. This does NOT change how arrays are passed in, so x is actually the same in both these subroutines:

```
/* Fortran */
SUBROUTINE BAR(arr)
  INTEGER :: arr(3)
  x=arr(1)
END SUBROUTINE
```

```
/* C/C++ */
FLINKAGE void FTN_NAME(BAR,bar) (int *arr) {
    x=arr[0];
}
```

- There is a subtle but important difference between the way f77 and f90 pass array arguments. f90 will pass an array object (which is not intelligible from C/C++) instead of a simple pointer if all of the following are true:
 - A f90 “INTERFACE” statement is available on the call side.
 - The subroutine is declared as taking an unspecified-length array (e.g., “myArr(:)”) or POINTER variable.

Because these f90 array objects can’t be used from C/C++, we recommend C/C++ routines either provide no f90 INTERFACE or else all the arrays in the INTERFACE are given explicit lengths.

- Multidimensional allocatable arrays are stored with the smallest index first in Fortran. C/C++ do not support allocatable multidimensional arrays, so they must fake them using arrays of pointers or index arithmetic.

```
/* Fortran */
SUBROUTINE BAR2(arr,len1,len2)
    INTEGER :: arr(len1,len2)
    INTEGER :: i,j
    DO j=1,len2
        DO i=1,len1
            arr(i,j)=i;
        END DO
    END DO
END SUBROUTINE
```

```
/* C/C++ */
FLINKAGE void FTN_NAME(BAR2,bar2) (int *arr,int *len1p,int *len2p) {
    int i,j; int len1=*len1p, len2=*len2p;
    for (j=0;j<len2;j++)
        for (i=0;i<len1;i++)
            arr[i+j*len1]=i;
}
```

- Fortran strings are passed in a very strange fashion. A string argument is passed as a character pointer and a length, but the length field, unlike all other Fortran arguments, is passed by value, and goes after all other arguments. Hence

```
/* Fortran */
SUBROUTINE CALL_BARS(arg)
    INTEGER :: arg
    CALL BARS('some string',arg);
END SUBROUTINE
```

```
/* C/C++ */
FLINKAGE void FTN_NAME(BARS,bars) (char *str,int *arg,int strlen) {
    char *s=(char *)malloc(strlen+1);
    memcpy(s,str,strlen);
    s[strlen]=0; /* nul-terminate string */
    printf("Received Fortran string '%s' (%d characters)\n",s,strlen);
    free(s);
}
```

- A f90 named TYPE can sometimes successfully be passed into a C/C++ struct, but this can fail if the compilers insert different amounts of padding. There does not seem to be a portable way to pass f90 POINTER variables

into C/C++, since different compilers represent POINTER variables differently.

6.10.10 How do I use Charm++ on Xeon Phi?

In general, no changes are required to use Charm++ on Xeon Phi. To compile code for Knights Landing, no special flags are required. To compile code for Knights Corner, one should build Charm++ with the `mic` option. In terms of network layers, we currently recommend building the MPI layer (`mpi-linux-x86_64`) except for machines with custom network layers, such as Cray systems, on which we recommend building for the custom layer (`gni-crayxc` for Cray XC machines, for example). To enable AVX-512 vector instructions, Charm++ can be built with `-xMIC-AVX512` on Intel compilers or `-mavx512f -mavx512er -mavx512cd -mavx512pf` for GNU compilers.

6.10.11 How do I use Charm++ on GPUs?

Charm++ users have two options when utilizing GPUs in Charm++.

The first is to write CUDA (or OpenCL, etc) code directly in their Charm++ applications. This does not take advantage of any of the special GPU-friendly features the Charm++ runtime provides and is similar to how programmers utilize GPUs in other parallel environments, e.g. MPI.

The second option is to leverage Charm++’s GPU library, GPU Manager. This library provides several useful features including:

- Automated data movement
- Ability to invoke callbacks at various points
- Host side pinned memory pooling
- Asynchronous kernel invocation
- Integrated tracing in Projections

To do this, Charm++ must be built with the `cuda` option. Users must describe their kernels using a work request struct, which includes the buffers to be copied, callbacks to be invoked, and kernel to be executed. Additionally, users can take advantage of a pre-allocated host side pinned memory pool allocated by the runtime via invoking `hapiPoolMalloc`. Finally, the user must compile this code using the appropriate `nvcc` compiler as per usual.

More details on using GPUs in Charm++ can be found in Section 2.3.14 of the Charm++ manual.

6.11 Converse Programming

6.11.1 What is Converse? Should I use it?

`Converse` is the low-level portable messaging layer that Charm++ is built on, but you don’t have to know anything about Converse to use Charm++. You might want to learn about Converse if you want a capable, portable foundation to implement a new parallel language on.

6.11.2 How much does getting a random number generator “right” matter?

`drand48` is nonportable and woefully inadequate for any real simulation task. Even if each processor seeds `drand48` differently, there is no guarantee that the streams of pseudo-random numbers won’t quickly overlap. A better generator would be required to “do it right” (See Park & Miller, CACM Oct. 88).

6.11.3 What should I use to get a proper random number generator?

Converse provides a 64-bit pseudorandom number generator based on the SPRNG package originally written by Ashok Shrinivasan at NCSA. For detailed documentation, please take a look at the Converse Extensions Manual on the Charm++ website. In short, you can use *CrnDrand()* function instead of the unportable *drand48()* in Charm++.

6.12 Charm++ and Converse Internals

6.12.1 How is the Charm++ source code organized and built?

All the Charm++ core source code is soft-linked into the `charm/<archname>/tmp` directory when you run the build script. The libraries and frameworks are under `charm/<archname>/tmp/libs`, in either `ck-libs` or `conv-libs`.

6.12.2 I just changed the Charm++ core. How do I recompile Charm++?

`cd` into the `charm/<archname>/tmp` directory and `make`. If you want to compile only a subset of the entire set of libraries, you can specify it to `make`. For example, to compile only the Charm++ RTS, type *make charm++*.

6.12.3 Do we have a *#define charm_version* somewhere? If not, which version number should I use for the current version?

Yes, there is a Charm++ version number defined in the macro `CHARM_VERSION`.

Threaded Charm++ (TCharm)

Contents

- *Threaded Charm++ (TCharm)*
 - *Motivation*
 - *Basic TCharm Programming*
 - * *Global Variables*
 - * *Input/Output*
 - * *Migration-Based Load Balancing*
 - *Advanced TCharm Programming*
 - * *Writing a Pup Routine*
 - * *Readonly Global Variables*
 - *Combining Frameworks*
 - *Command-line Options*
 - *Writing a library using TCharm*

7.1 Motivation

Charm++ includes several application frameworks, such as the Finite Element Framework, the Multiblock Framework, and AMPI. These frameworks do almost all their work in load balanced, migratable threads.

The Threaded Charm++ Framework, TCharm, provides both common runtime support for these threads and facilities for combining multiple frameworks within a single program. For example, you can use TCharm to create a Finite Element Framework application that also uses AMPI to communicate between Finite Element chunks.

Specifically, TCharm provides language-neutral interfaces for:

1. Program startup, including read-only global data setup and the configuration of multiple frameworks.
2. Run-time load balancing, including migration.
3. Program shutdown.

The first portion of this manual describes the general properties of TCharm common to all the application frameworks, such as program contexts and how to write migratable code. The second portion describes in detail how to combine separate frameworks into a single application.

7.2 Basic TCharm Programming

Any routine in a TCharm program runs in one of two contexts:

Serial Context Routines that run on only one processor and with only one set of data. There are absolutely no limitations on what a serial context routine can do—it is as if the code were running in an ordinary serial program. Startup and shutdown routines usually run in the serial context.

Parallel Context Routines that run on several processors, and may run with several different sets of data on a single processor. This kind of routine must obey certain restrictions. The program’s main computation routines always run in the parallel context.

Parallel context routines run in a migratable, user-level thread maintained by TCharm. Since there are normally several of these threads per processor, any code that runs in the parallel context has to be thread-safe. However, TCharm is non-preemptive, so it will only switch threads when you make a blocking call, like “MPI_Recv” or “FEM_Update_field”.

7.2.1 Global Variables

By “global variables”, we mean anything that is stored at a fixed, preallocated location in memory. In C, this means variables declared at file scope or with the static keyword. In Fortran, this is either variables that are part of a COMMON block, declared inside a MODULE or variables with the SAVE attribute.

Global variables are shared by all the threads on a processor, which makes using global variables extremely error prone. To see why this is a problem, consider a program fragment like:

```
foo=a
call MPI_Recv(...)
b=foo
```

After this code executes, we might expect b to always be equal to a. but if foo is a global variable, MPI_Recv may block and foo could be changed by another thread.

For example, if two threads execute this program, they could interleave like:

<i>Thread 1</i>	<i>Thread 2</i>
foo=1	
block in MPI_Recv	
	foo=2
	block in MPI_Recv
b=foo	

At this point, thread 1 might expect b to be 1; but it will actually be 2. From the point of view of thread 1, the global variable foo suddenly changed its value during the call to MPI_Recv.

There are several possible solutions to this problem:

- Never use global variables—only use parameters or local variables. This is the safest and most general solution. One standard practice is to collect all the globals into a C struct or Fortran type named “Globals”, and pass a pointer to this object to all your subroutines. This also combines well with the pup method for doing migration-based load balancing, as described in Section 7.3.1.
- Never write *different* values to global variables. If every thread writes the same value, global variables can be used safely. For example, you might store some parameters read from a configuration file like the simulation timestep Δt . See Section 7.3.2 for another, more convenient way to set such variables.
- Never issue a blocking call while your global variables are set. This will not work on a SMP version of Charm++, where several processors may share a single set of global variables. Even on a non-SMP version, this is a dangerous solution, because someday someone might add a blocking call while the variables are set. This is only a reasonable solution when calling legacy code or using old serial libraries that might use global variables.

The above only applies to routines that run in the parallel context. There are no restrictions on global variables for serial context code.

7.2.2 Input/Output

In the parallel context, there are several limitations on open files. First, several threads may run on one processor, so Fortran Logical Unit Numbers are shared by all the threads on a processor. Second, open files are left behind when a thread migrates to another processor—it is a crashing error to open a file, migrate, then try to read from the file.

Because of these restrictions, it is best to open files only when needed, and close them as soon as possible. In particular, it is best if there are no open files whenever you make blocking calls.

7.2.3 Migration-Based Load Balancing

The Charm++ runtime framework includes an automatic run-time load balancer, which can monitor the performance of your parallel program. If needed, the load balancer can “migrate” threads from heavily-loaded processors to more lightly-loaded processors, improving the load balance and speeding up the program. For this to be useful, you need to pass the link-time argument `-balancer B` to set the load balancing algorithm, and the run-time argument `+vp N` (use `N` virtual processors) to set the number of threads. The ideal number of threads per processor depends on the problem, but we’ve found five to a hundred threads per processor to be a useful range.

When a thread migrates, all its data must be brought with it. “Stack data”, such as variables declared locally in a subroutine, will be brought along with the thread automatically. Global data, as described in Section 7.2.1, is never brought with the thread and should generally be avoided.

“Heap data” in C is structures and arrays allocated using `malloc` or `new`; in Fortran, heap data is `TYPE`s or arrays allocated using `ALLOCATE`. To bring heap data along with a migrating thread, you have two choices: write a pup routine or use `isomalloc`. Pup routines are described in Section 7.3.1.

Isomalloc is a special mode which controls the allocation of heap data. You enable `isomalloc` allocation using the link-time flag “-memory isomalloc”. With `isomalloc`, migration is completely transparent—all your allocated data is automatically brought to the new processor. The data will be unpacked at the same location (the same virtual addresses) as it was stored originally; so even cross-linked data structures that contain pointers still work properly.

The limitations of `isomalloc` are:

- Wasted memory. `Isomalloc` uses a special interface¹ to acquire memory, and the finest granularity that can be acquired is one page, typically 4KB. This means if you allocate a 2-entry array, `isomalloc` will waste an entire 4KB page. We should eventually be able to reduce this overhead for small allocations.

¹ The interface used is `mmap`.

- Limited space on 32-bit machines. Machines where pointers are 32 bits long can address just 4GB (2^{32} bytes) of virtual address space. Additionally, the operating system and conventional heap already use a significant amount of this space; so the total virtual address space available is typically under 1GB. With isomalloc, all processors share this space, so with just 20 processors the amount of memory per processor is limited to under 50MB! This is an inherent limitation of 32-bit machines; to run on more than a few processors you must use 64-bit machines or avoid isomalloc.

7.3 Advanced TCharm Programming

The preceding features are enough to write simple programs that use TCharm-based frameworks. These more advanced techniques provide the user with additional capabilities or flexibility.

7.3.1 Writing a Pup Routine

The runtime system can automatically move your thread stack to the new processor, but unless you use isomalloc, you must write a pup routine to move any global or heap-allocated data to the new processor. A pup (Pack/UnPack) routine can perform both packing (converting your data into a network message) and unpacking (converting the message back into your data). A pup routine is passed a pointer to your data block and a special handle called a “pupper”, which contains the network message.

In a pup routine, you pass all your heap data to routines named `pup_type` or `fpup_type`, where `type` is either a basic type (such as `int`, `char`, `float`, or `double`) or an array type (as before, but with a “s” suffix). Depending on the direction of packing, the pupper will either read from or write to the values you pass- normally, you shouldn’t even know which. The only time you need to know the direction is when you are leaving a processor, or just arriving. Correspondingly, the pupper passed to you may be deleting (indicating that you are leaving the processor, and should delete your heap storage after packing), unpacking (indicating you’ve just arrived on a processor, and should allocate your heap storage before unpacking), or neither (indicating the system is merely sizing a buffer, or checkpointing your values).

pup functions are much easier to write than explain- a simple C heap block and the corresponding pup function is:

```
typedef struct {
    int n1; /*Length of first array below*/
    int n2; /*Length of second array below*/
    double *arr1; /*Some doubles, allocated on the heap*/
    int *arr2; /*Some ints, allocated on the heap*/
} my_block;

void pup_my_block(pup_er p, my_block *m)
{
    if (pup_isUnpacking(p)) { /*Arriving on new processor*/
        m->arr1=malloc(m->n1*sizeof(double));
        m->arr2=malloc(m->n2*sizeof(int));
    }
    pup_doubles(p, m->arr1, m->n1);
    pup_ints(p, m->arr2, m->n2);
    if (pup_isDeleting(p)) { /*Leaving old processor*/
        free(m->arr1);
        free(m->arr2);
    }
}
```

This single pup function can be used to copy the `my_block` data into a message buffer and free the old heap storage (deleting pupper); allocate storage on the new processor and copy the message data back (unpacking pupper); or save the heap data for debugging or checkpointing.

A Fortran block TYPE and corresponding pup routine is as follows:

```

MODULE my_block_mod
  TYPE my_block
    INTEGER :: n1,n2x,n2y
    DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:) :: arr1
    INTEGER, ALLOCATABLE, DIMENSION(:,) :: arr2
  END TYPE
END MODULE

SUBROUTINE pup_my_block(p,m)
  IMPLICIT NONE
  USE my_block_mod
  USE pupmod
  INTEGER :: p
  TYPE(my_block) :: m
  IF (fpup_isUnpacking(p)) THEN
    ALLOCATE(m%arr1(m%n1))
    ALLOCATE(m%arr2(m%n2x,m%n2y))
  END IF
  call fpup_doubles(p,m%arr1,m%n1)
  call fpup_ints(p,m%arr2,m%n2x*m%n2y)
  IF (fpup_isDeleting(p)) THEN
    DEALLOCATE(m%arr1)
    DEALLOCATE(m%arr2)
  END IF
END SUBROUTINE

```

You indicate to TCharm that you want a pup routine called using the routine below. An arbitrary number of blocks can be registered in this fashion.

```
void TCHARM_Register(void *block, TCharmPupFn pup_fn)
```

```

SUBROUTINE TCHARM_Register(block,pup_fn)
  TYPE(varies), POINTER :: block
  SUBROUTINE :: pup_fn

```

Associate the given data block and pup function. Can only be called from the parallel context. For the declarations above, you call TCHARM_Register as:

```

/*In C/C++ driver() function*/
my_block m;
TCHARM_Register(m, (TCharmPupFn)pup_my_block);

```

```

!- In Fortran driver subroutine
use my_block_mod
interface
  subroutine pup_my_block(p,m)
    use my_block_mod
    INTEGER :: p
    TYPE(my_block) :: m
  end subroutine
end interface
TYPE(my_block), TARGET :: m
call TCHARM_Register(m,pup_my_block)

```

Note that the data block must be allocated on the stack. Also, in Fortran, the “TARGET” attribute must be used on the block (as above) or else the compiler may not update values during a migration, because it believes only it can access

the block.

```
void TCHARM_Migrate()
```

```
subroutine TCHARM_Migrate()
```

Informs the load balancing system that you are ready to be migrated, if needed. If the system decides to migrate you, the pup function passed to TCHARM_Register will first be called with a sizing pupper, then a packing, deleting pupper. Your stack and pupped data will then be sent to the destination machine, where your pup function will be called with an unpacking pupper. TCHARM_Migrate will then return. Can only be called from in the parallel context.

7.3.2 Readonly Global Variables

You can also use a pup routine to set up initial values for global variables on all processors. This pup routine is called with only a pup handle, just after the serial setup routine, and just before any parallel context routines start. The pup routine is never called with a deleting pup handle, so you need not handle that case.

A C example is:

```
int g_arr[17];
double g_f;
int g_n; /*Length of array below*/
float *g_allocated; /*heap-allocated array*/

void pup_my_globals(pup_er p)
{
    pup_ints(p,g_arr,17);
    pup_double(p,&g_f);
    pup_int(p,&g_n);
    if (pup_isUnpacking(p)) { /*Arriving on new processor*/
        g_allocated=malloc(g_n*sizeof(float));
    }
    pup_floats(p,g_allocated,g_n);
}
```

A Fortran example is:

```
MODULE my_globals_mod
    INTEGER :: g_arr(17)
    DOUBLE PRECISION :: g_f
    INTEGER :: g_n
    SINGLE PRECISION, ALLOCATABLE :: g_allocated(:)
END MODULE

SUBROUTINE pup_my_globals(p)
    IMPLICIT NONE
    USE my_globals_mod
    USE pupmod
    INTEGER :: p
    call fpup_ints(p,g_arr,17)
    call fpup_double(p,g_f)
    call fpup_int(p,g_n)
    IF (fpup_isUnpacking(p)) THEN
        ALLOCATE(g_allocated(g_n))
    END IF
    call fpup_floats(p,g_allocated,g_n)
END SUBROUTINE
```


You register your global variable pup routine using the method below. Multiple pup routines can be registered the same way.

```
void TCHARM_Readonly_globals(TCharmPupGlobalFn pup_fn)
```

```
SUBROUTINE TCHARM_Readonly_globals(pup_fn)
SUBROUTINE :: pup_fn
```

7.4 Combining Frameworks

This section describes how to combine multiple frameworks in a single application. You might want to do this, for example, to use AMPI communication inside a finite element method solver.

You specify how you want the frameworks to be combined by writing a special setup routine that runs when the program starts. The setup routine must be named `TCHARM_User_setup`. If you declare a user setup routine, the standard framework setup routines (such as the FEM framework's init routine) are bypassed, and you do all the setup in the user setup routine.

The setup routine creates a set of threads and then attaches frameworks to the threads. Several different frameworks can be attached to one thread set, and there can be several sets of threads; however, the most frameworks cannot be attached more than once to single set of threads. That is, a single thread cannot have two attached AMPI frameworks, since the `MPI_COMM_WORLD` for such a thread would be indeterminate.

```
void TCHARM_Create(int nThreads, TCharmThreadStartFn thread_fn)
```

```
SUBROUTINE TCHARM_Create(nThreads, thread_fn)
INTEGER, INTENT(in) :: nThreads
SUBROUTINE :: thread_fn
```

Create a new set of TCharm threads of the given size. The threads will execute the given function, which is normally your user code. You should call `TCHARM_Get_num_chunks()` to get the number of threads from the command line. This routine can only be called from your `TCHARM_User_setup` routine.

You then attach frameworks to the new threads. The order in which frameworks are attached is irrelevant, but attach commands always apply to the current set of threads.

To attach a chare array to the TCharm array, use:

```
CkArrayOptions TCHARM_Attach_start(CkArrayID *retTCharmArray, int
*retNumElts)
```

This function returns a `CkArrayOptions` object that will bind your chare array to the TCharm array, in addition to returning the TCharm array proxy and number of elements by reference. If you are using frameworks like AMPI, they will automatically attach themselves to the TCharm array in their initialization routines.

7.5 Command-line Options

The complete set of link-time arguments relevant to TCharm is:

- memory isomalloc** Enable memory allocation that will automatically migrate with the thread, as described in Section 7.2.3.
- balancer B** Enable this load balancing strategy. The current set of balancers B includes `RefineLB` (make only small changes each time), `MetisLB` (remap threads using graph partitioning library), `HeapCentLB` (remap threads

using a greedy algorithm), and RandCentLB (remap threads to random processors). You can only have one balancer.

-module F Link in this framework. The current set of frameworks F includes ampi, collide, fem, mblock, and netfem. You can link in multiple frameworks.

The complete set of command-line arguments relevant to TCharm is:

+p N Run on N physical processors.

+vp N Create N “virtual processors”, or threads. This is the value returned by TCharmGetNumChunks.

++debug Start each program in a debugger window. See Charm++ Installation and Usage Manual for details.

+tcharm_stacksize N Create N-byte thread stacks. This value can be overridden using TCharmSetStackSize().

+tcharm_nomig Disable thread migration. This can help determine whether a problem you encounter is caused by our migration framework.

+tcharm_nothread Disable threads entirely. This can help determine whether a problem you encounter is caused by our threading framework. This generally only works properly when using only one thread.

+tcharm_trace F Trace all calls made to the framework F. This can help to understand a complex program. This feature is not available if Charm++ was compiled with CMK_OPTIMIZE.

7.6 Writing a library using TCharm

Until now, things were presented from the perspective of a user—one who writes a program for a library written on TCharm. This section gives an overview of how to go about writing a library in Charm++ that uses TCharm.

- Compared to using plain MPI, TCharm provides the ability to access all of Charm++, including arrays and groups.
- Compared to using plain Charm++, using TCharm with your library automatically provides your users with a clean C/F90 API (described in the preceding chapters) for basic thread memory management, I/O, and migration. It also allows you to use a convenient “thread->suspend()” and “thread->resume()” API for blocking a thread, and works properly with the load balancer, unlike CthSuspend/CthAwaken.

The overall scheme for writing a TCharm-based library “Foo” is:

1. You must provide a FOO_Init routine that creates anything you’ll need, which normally includes a Chare Array of your own objects. The user will call your FOO_Init routine from their main work routine; and normally FOO_Init routines are collective.
2. In your FOO_Init routine, create your array bound it to the running TCharm threads, by creating it using the CkArrayOptions returned by TCHARM_Attach_start. Be sure to only create the array once, by checking if you’re the master before creating the array.

One simple way to make the non-master threads block until the corresponding local array element is created is to use TCharm semaphores. These are simply a one-pointer slot you can assign using TCharm::semaPut and read with TCharm::semaGet. They’re useful in this context because a TCharm::semaGet blocks if a local TCharm::semaGet hasn’t yet executed.

```
//This is either called by FooFallbackSetuo mentioned above, or by the user
//directly from TCHARM_User_setup (for multi-module programs)
void FOO_Init(void)
{
    if (TCHARM_Element()==0) {
        CkArrayID threadsAID; int nchunks;
        CkArrayOptions opts=TCHARM_Attach_start(&threadsAID,&nchunks);
```

(continues on next page)

(continued from previous page)

```

//actually create your library array here (FooChunk in this case)
    CkArrayID aid = CProxy_FooChunk::ckNew(opt);
}
FooChunk *arr=(FooChunk *)TCharm::semaGet(FOO_TCHARM_SEMAID);
}

```

3. Depending on your library API, you may have to set up a thread-private variable(Ctv) to point to your library object. This is needed to regain context when you are called by the user. A better design is to avoid the Ctv, and instead hand the user an opaque handle that includes your array proxy.

```

//_fooptr is the Ctv that points to the current chunk FooChunk and is only valid_
↪in
//routines called from fooDriver()
CtvStaticDeclare(FooChunk *, _fooptr);

/* The following routine is listed as an initcall in the .ci file */
/*initcall*/ void fooNodeInit(void)
{
    CtvInitialize(FooChunk*, _fooptr);
}

```

4. Define the array used by the library

```

class FooChunk: public TCharmClient1D {
    CProxy_FooChunk thisProxy;
protected:
    //called by TCharmClient1D when thread changes
    virtual void setupThreadPrivate(CthThread forThread)
    {
        CtvAccessOther(forThread, _fooptr) = this;
    }

    FooChunk(CkArrayID aid):TCharmClient1D(aid)
    {
        thisProxy = this;
        tCharmClientInit();
        TCharm::semaPut(FOO_TCHARM_SEMAID,this);
        //add any other initialization here
    }

    virtual void pup(PUP::er &p) {
        TCharmClient1D::pup(p);
        //usual pup calls
    }

    // ...any other calls you need...
    int doCommunicate(...);
    void recvReply(someReplyMsg *m);
    .....
}

```

5. Block a thread for communication using thread->suspend and thread->resume

```

int FooChunk::doCommunicate(...)
{

```

(continues on next page)

(continued from previous page)

```

    replyGoesHere = NULL;
    thisProxy[destChunk].sendRequest(...);
    thread->suspend(); //wait for reply to come back
    return replyGoesHere->data;
}

void FooChunk::recvReply(someReplyMsg *m)
{
    if(replyGoesHere!=NULL) CkAbort("FooChunk: unexpected reply\n");
    replyGoesHere = m;
    thread->resume(); //Got the reply -- start client again
}

```

6. Add API calls. This is how user code running in the thread interacts with the newly created library. Calls to TCHARM_API_TRACE macro must be added to the start of every user-callable method. In addition to tracing, these disable isomalloc allocation.

The charm-api.h macros CLINKAGE, FLINKAGE and FTN_NAME should be used to provide both C and FORTRAN versions of each API call. You should use the “MPI capitalization standard”, where the library name is all caps, followed by a capitalized first word, with all subsequent words lowercase, separated by underscores. This capitalization system is consistent, and works well with case-insensitive languages like Fortran.

Fortran parameter passing is a bit of an art, but basically for simple types like int (INTEGER in fortran), float (SINGLE PRECISION or REAL*4), and double (DOUBLE PRECISION or REAL*8), things work well. Single parameters are always passed via pointer in Fortran, as are arrays. Even though Fortran indexes arrays based at 1, it will pass you a pointer to the first element, so you can use the regular C indexing. The only time Fortran indexing need be considered is when the user passes you an index-the int index will need to be decremented before use, or incremented before a return.

```

CLINKAGE void FOO_Communicate(int x, double y, int * arr) {
    TCHARM_API_TRACE("FOO_Communicate", "foo"); //2nd parameter is the name of the_
    →library
    FooChunk *f = CtvAccess(_fooptr);
    f->doCommunicate(x, y, arr);
}

//In fortran, everything is passed via pointers
FLINKAGE void FTN_NAME(FOO_COMMUNICATE, foo_communicate)
    (int *x, double *y, int *arr)
{
    TCHARM_API_TRACE("FOO_COMMUNICATE", "foo");
    FooChunk *f = CtvAccess(_fooptr);
    f->doCommunicate(*x, *y, arr);
}

```

Finite Element Method (FEM) Framework

Contents

- *Finite Element Method (FEM) Framework*
 - *Introduction*
 - * *Philosophy*
 - * *Terminology*
 - * *Structure of a Classic FEM Framework Program*
 - * *Structure of an AMPI FEM Framework Program*
 - * *Compilation and Execution*
 - *FEM Framework API Reference*
 - * *Utility*
 - *Mesh Nodes and Elements*
 - * *Mesh Entity Types*
 - * *Mesh Entity Manipulation*
 - * *Entity Inquiry*
 - * *Advanced Entity Manipulation*
 - *Meshes*
 - * *Mesh Routines*
 - * *Mesh Utility*
 - * *Advanced Mesh Manipulation*
 - *Mesh Ghosts*

- * *Ghost Numbering*
- * *Setting up the ghost layer*
- * *Symmetries and Ghosts-Geometric Layer*
- * *Advanced Symmetries and Ghosts-Lower Layer*
- *Older Mesh Routines*
 - * *Old Mesh Data*
 - * *Old Ghost Numbering*
 - * *Old Backward Compatibility*
 - * *Old Sparse Data*
- *Mesh Modification*
- *IDXL Communication*
 - * *Index Lists*
 - * *Data Layout*
 - * *IDXL Communication*
- *Old Communication Routines*
 - * *Ghost Communication*
 - * *Ghost List Exchange*
- *ParFUM*
 - * *Adaptivity Initialization*
 - * *Preparing the Mesh for Adaptivity*
 - * *Modifying the Mesh*
 - * *Verify correctness of the Mesh*
 - * *ParFUM developers*

8.1 Introduction

The Finite Element Method (FEM) approach is used in many engineering applications with irregular domains, from elastic deformation problems to crack propagation to fluid flow. Charm++ is a free message-passing parallel runtime system for machines from clusters of workstations to tightly-coupled SMPs. The Charm++ FEM framework allows you to write a parallel FEM program, in C or Fortran 90, that closely resembles a serial version but includes a few framework calls.

Using the FEM framework also allows you to take advantage of all the features of Charm++, including run-time load balancing, performance monitoring and visualization, and checkpoint/restart, with no additional effort. The FEM framework also combines naturally with other Charm++ frameworks built on TCHARM.

The FEM framework has been undergoing a wave of recent improvements. A choice to rename the new version ParFUM has been adopted. ParFUM is short for Parallel Framework for Unstructured Meshes. Section 8.10 describes some of the new features included in ParFUM that were not present in FEM.

8.1.1 Philosophy

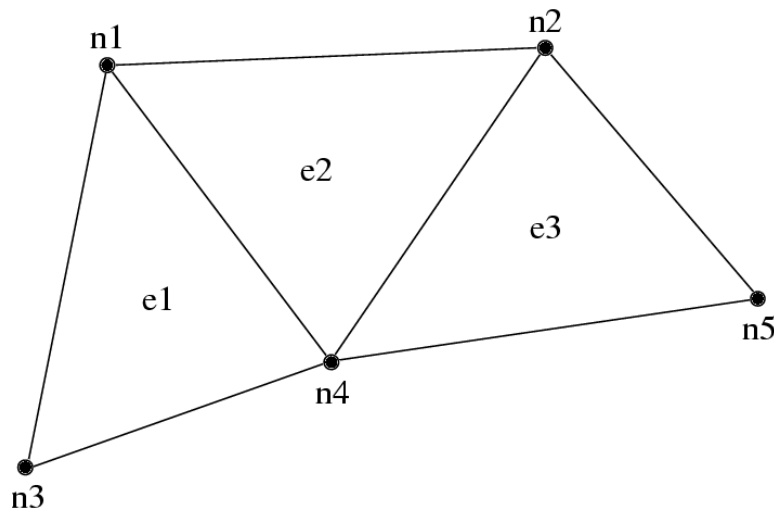
The Charm++ FEM framework is designed to be flexible, in that it provided a few very general operations, such as loading and partitioning a “mesh.” In describing these operations, we draw on examples from structural analysis, but in fact the same calls can be used for other applications, including fluid dynamics or partial differential equations solvers, or even general-purpose graph manipulation.

For example, the FEM framework does not specify the number of spatial dimensions. Node locations are treated as just another kind of node data, with no restrictions on the number of data items. This allows the FEM framework to work with problems having any number of spatial dimensions.

8.1.2 Terminology

A FEM program manipulates elements and nodes. An **element** is a portion of the problem domain, also known as a cell, and is typically some simple shape like a triangle, square, or hexagon in 2D; or tetrahedron or rectangular solid in 3D. A **node** is a point in the domain, and is often the vertex of several elements. Together, the elements and nodes form a **mesh**, which is the central data structure in the FEM framework.

An element knows which nodes surround it via the element’s **connectivity table**, which lists the nodes adjacent to each element.



6: 3-element, 5 node mesh.

5: Connectivity table for mesh in figure 6.

Element	Adjacent Nodes		
e1	n1	n3	n4
e2	n1	n2	n4
e3	n2	n4	n5

A typical FEM program performs some element-by-element calculations which update adjacent node values; then some node-by-node calculations. For example, a material dynamics program has the structure:

```
time loop
  element loop-- Element deformation applies forces to
    surrounding nodes
  node loop-- Forces and boundary conditions change node
```

(continues on next page)

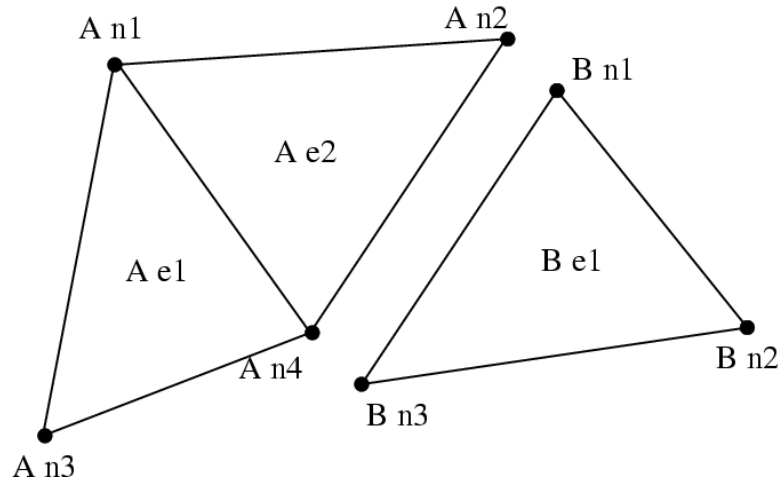
(continued from previous page)

```

    positions
end time loop

```

We can parallelize such FEM programs by partitioning the serial mesh elements into several smaller meshes, or **chunks**. There is normally at least one chunk per processor; and often even more. During partitioning, we give nodes and elements new, **local** numbers within that chunk. In the figure below, we have partitioned the mesh above into two chunks, A and B.



7: Partitioned mesh.

6: Connectivity table for chunk A in figure 7.

Element	Adjacent Nodes		
e1	n1	n3	n4
e2	n1	n2	n4

7: Connectivity table for chunk B in figure 7.

Element	Adjacent Nodes		
e1	n1	n2	n3

Note that chunk A's node n2 and B's node n1 were actually the same node in the original mesh- partitioning split this single node into two shared copies (one on each chunk). However, since adding forces is associative, we can handle shared nodes by computing the forces normally (ignoring the existence of the other chunk), then adding both chunks' net force for the shared node together. This "node update" will give us the same resulting force on each shared node as we would get without partitioning, thus the same positions, thus the same final result.

For example, under hydrostatic pressure, each chunk might compute a local net force vector for its nodes as shown in Figure 8 (a). After adding forces across chunks, we have the consistent global forces shown in Figure 8 (b).

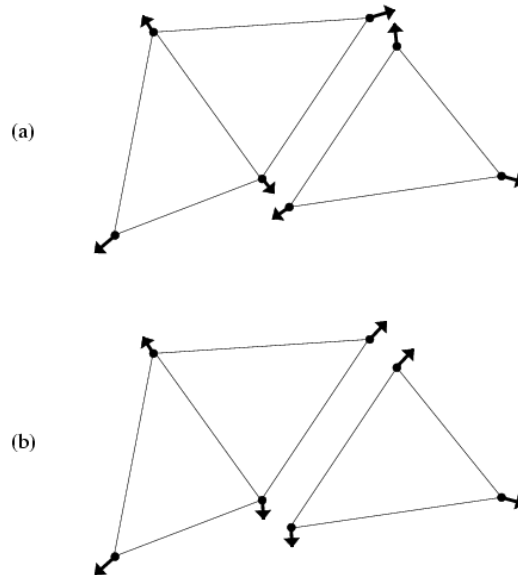
Hence, each chunk's time loop has the structure:

```

chunk time loop
  element loop-- Element deformation applies forces to
  surrounding nodes
  <update forces on shared nodes>
  node loop-- Forces and boundary conditions change node

```

(continues on next page)



8: A force calculation decomposed across chunks: (a) before update (b) after updating forces across nodes.

(continued from previous page)

```
positions
end time loop
```

This is exactly the form of the time loop for a Charm++ FEM framework program. The framework will accept a serial mesh, partition it, distribute the chunks to each processor, then you run your time loop to perform analysis and communication.

8.1.3 Structure of a Classic FEM Framework Program

A classic FEM framework program consists of two subroutines: `init()` and `driver()`. `init()` is called by the FEM framework only on the first processor - this routine typically does specialized I/O, startup and shutdown tasks. `driver()` is called for every chunk on every processor, and does the main work of the program. In the language of the TCHARM manual, `init()` runs in the serial context, and `driver()` runs in the parallel context.

```
subroutine init
  read the serial mesh and configuration data
end subroutine
/* after init, the FEM framework partitions the mesh */
subroutine driver
  get local mesh chunk
  time loop
    FEM computations
    communicate boundary conditions
    more FEM computations
  end time loop
end subroutine
```

In this mode, the FEM framework sets up a default writing mesh during `init()`, partitions the mesh after `init()`, and sets up the partitioned mesh as the default reading mesh during `driver()`.

8.1.4 Structure of an AMPI FEM Framework Program

In addition to the classic init/driver structure above, you can write an FEM framework program using the MPI style. This is a more general, more flexible method of running the program, but it is more complicated than the classic mode. All FEM framework calls are available in either mode.

```
main program
  MPI_Init
  FEM_Init (MPI_COMM_WORLD)
  if (I am master processor)
    read mesh
  partition mesh
  time loop
    FEM computations
    communicate boundary conditions
    more FEM computations
  end time loop
end main program
```

In this mode, the FEM framework does not set a default reading or writing mesh, and does no partitioning; so you must use the FEM_Mesh routines to create and partition your mesh. See the AMPI manual for details on how to declare the main routine.

The driver() portion of a classic FEM program strongly resembles an MPI mode main routine—in fact, a classic FEM program can even make MPI calls from its driver() routine, because the FEM framework is implemented directly on top of MPI.

There is even a special shell script for collecting up the FEM framework source code to build a non-Charm, MPI-only version of the FEM framework. To build FEM in this manner, you first build Charm++ normally, then run a script to collect up the necessary source files (the FEM framework, a small number of Charm configuration and utility files, and the METIS library), and finally build the library using the usual MPI compiler commands:

```
$ cd charm/
$ ./src/libs/ck-libs/fem/make_fem_alone.sh
$ cd fem_alone/
$ mpicc -I. -DFEM_ALONE=1 -c *.c *.C
$ ar cr libfem_alone.a *.o
```

You will then have to build your application with the MPI compilers, and manually point to this “fem_alone” directory to find include files and the new FEM library. A typical compiler invocation would be:

```
$ mpif90 -I$HOME/charm/fem_alone -L$HOME/charm/fem_alone foo.f90 -lfem_alone -o foo
```

This “standalone”, non-Charm++ method of building the FEM framework prevents the use of load balancing or the other features of Charm++, so we do not recommend it for normal use.

8.1.5 Compilation and Execution

A FEM framework program is a Charm++ program, so you must begin by downloading the latest source version of Charm++ from <http://charm.cs.uiuc.edu/>. Build the source with `./build FEM version` or `cd` into the build directory, `version/tmp`, and type `make FEM`. To compile a FEM program, pass the `-language fem` (for C) or `-language femf` (for Fortran) option to `charmcc`. You can also build using the “fem_alone” mode described at the end of the section above.

In a charm installation, see `charm/version/pgms/charm++/fem/` for several example and test programs.

At runtime, a Charm++/FEM framework program accepts the following options, in addition to all the usual Charm++ options described in the Charm++ “Installation and Usage Manual”.

- `+vp v`
Create v mesh chunks, or “virtual processors”. By default, the number of mesh chunks is equal to the number of physical processors (set with `+p p`).
- `-write`
Skip `driver()`. After running `init()` normally, the framework partitions the mesh, writes the mesh partitions to files, and exits. As usual, the `+vp v` option controls the number of mesh partitions.

This option is only used in the classic mode—MPI-style programs are not affected.
- `-read`
Skip `init()`. The framework reads the partitioned input mesh from files and calls `driver()`. Together with `-write`, this option allows you to separate out the mesh preparation and partitioning phase from the actual parallel solution run.

This can be useful, for example, if `init()` requires more memory to hold the unpartitioned mesh than is available on one processor of the parallel machine. To avoid this limitation, you can run the program with `-write` on a machine with a lot of memory to prepare the input files, then copy the files and run with `-read` on a machine with a lot of processors.

`-read` can also be useful during debugging or performance tuning, by skipping the (potentially slow) mesh preparation phase. This option is only used in the classic mode—MPI-style programs are not affected.
- `+tcharm_trace fem`
Give a diagnostic printout on every call into the FEM framework. This can be useful for locating a sudden crash, or understanding how the program and framework interact. Because printing the diagnostics can slow a program down, use this option with care.

8.2 FEM Framework API Reference

Some of the routines in the FEM framework have different requirements or meanings depending on where they are called from. When a routine is described as being “called from driver”, this means it is called in the parallel context—from `driver()` itself, any subroutine called by `driver()`, or from whatever routine is run by the FEM-attached TCHARM threads. When a routine is described as being “called from init”, this means it is called in the serial context—from `init()` itself, from any subroutine called from `init()`, from a routine called by `FEM_Update_mesh`, or from whatever TCHARM code executes before the `FEM_Attach`.

8.2.1 Utility

```
int FEM_Num_partitions();
```

```
INTEGER FUNCTION :: FEM_Num_partitions()
```

Return the number of mesh chunks in the current computation. Can only be called from the driver routine.

```
int FEM_My_partition();
```

```
INTEGER FUNCTION :: FEM_My_partition()
```

Return the number of the current chunk, from 0 to `num_partitions-1`. Can only be called from the driver routine.

```
double FEM_Timer();
```

```
DOUBLE PRECISION FUNCTION :: FEM_Timer()
```

Return the current wall clock time, in seconds. Resolution is machine-dependent, but is at worst 10ms.

```
void FEM_Print_partition();
```

```
SUBROUTINE FEM_Print_partition()
```

Print a debugging representation of the current chunk’s mesh. Prints the entire connectivity array, and data associated with each local node and element.

```
void FEM_Print(const char *str);
```

```
SUBROUTINE FEM_Print(str)
CHARACTER*, INTENT(IN) :: str
```

Print the given string, with “[<chunk number>]” printed before the text.

This routine is no longer required: you can now use the usual printf, PRINT, or WRITE statements.

8.3 Mesh Nodes and Elements

These routines describe and retrieve the finite element mesh for this computation. A **mesh**, from the framework’s perspective, is a list of elements, nodes, and other data that describes the computational domain. The FEM framework provides extensive support for creating, manipulating, and partitioning meshes.

A **serial mesh** consists of a single large piece. It’s usually easiest to read and write serial meshes to existing, non-parallel file formats, and it can be easier to manipulate serial meshes. By contrast, a **parallel mesh** consists of several pieces, called **chunks** or partitions. Different processors can work on different pieces of a parallel mesh, so most of the computation is done using parallel meshes. A simple program might create or read in a single serial mesh in init, get a local chunk of the partitioned¹ mesh in driver, and work on that chunk for the rest of the program. A more complex program might set an initial mesh in init; then get, work on, reassemble and repartition the mesh several times in driver via FEM_Update_mesh.

8.3.1 Mesh Entity Types

A mesh consists of **entities**, such as nodes and elements. Entities always have a **local number**, which is just the entities’ current index in its array. Entities may also have a **global number**, which is the entity’s index in the unpartitioned serial mesh. Entities have data values called **attributes**. For example, the location of each node might be called the “location” attribute of the “node” entity type. Attributes are always stored in regular arrays indexed by the entity’s local number. This table lists the different attributes that can be read or written for each type of entity.

A **shared entity** is a boundary entity that two or more chunks can both update—currently, only nodes can be shared. Shared nodes are mixed in with regular nodes, and the framework currently provides no way to identify which nodes are shared.

A **ghost entity** is a boundary entity that is asymmetrically shared—one side provides values for the ghost from one of its real entities, and the other sides accept read-only copies of these values. Ghosts are described in more detail in Section 8.5, and can be accessed by adding the constant FEM_GHOST to the corresponding real entity’s type.

¹ The framework uses the excellent graph partitioning package Metis.

The different kinds of entities are described in the following sections.

Real Entity	Ghost Entity
FEM_NODE	FEM_GHOST+FEM_NODE
FEM_ELEM+ <i>elType</i>	FEM_GHOST+FEM_ELEM+ <i>elType</i>
FEM_SPARSE+ <i>sparseType</i>	FEM_GHOST+FEM_SPARSE+ <i>sparseType</i>

Nodes

FEM_NODE is the entity code for nodes, the simplest kind of entity. A node is a single point in the domain, and elements are defined by their nodes. Nodes can have the following attributes:

- FEM_DATA+*tag* Uninterpreted user data, which might include material properties, boundary conditions, flags, etc. User data can have any data type and width. *tag* can be any number from 0 to one billion—it allows you to register several data fields with a single entity.
- FEM_GLOBALNO Global node numbers. Always a 1-wide index type.
- FEM_SYMMETRIES Symmetries that apply to this node. Always a 1-wide FEM_BYTE.
- FEM_NODE_PRIMARY Marker indicating that this chunk is responsible for this node. Every node is primary in exactly one chunk. This attribute is always a 1-wide FEM_BYTE containing 0 or 1.

Elements

FEM_ELEM+*elType* is the entity code for one kind of element. *elType* is a small, user-defined value that uniquely identifies this element type. Like nodes, elements can have the attributes FEM_DATA+*tag*, FEM_GLOBALNO, or FEM_SYMMETRIES; but every element type must have this attribute:

- FEM_CONN Lists the numbers of the nodes around this element. See the description in the ghost section for special ghost connectivity. Always an index type-FEM_INDEX_0 for C-style 0-based node indexing, or FEM_INDEX_1 for Fortran-style 1-based node indexing.

Sparse Elements

FEM_SPARSE+*sparseType* is the entity code for one kind of sparse element. Again, *sparseType* is a small, user-defined unique value. The only difference between ordinary elements and sparse elements regards partitioning. Ignoring ghosts, ordinary elements are never duplicated—each element is sent to its own chunk. Sparse elements may be duplicated, and are always dependent on some other entity for their partitioning. Sparse elements have all the attributes of ordinary elements: FEM_DATA+*tag*, FEM_GLOBALNO, FEM_SYMMETRIES, and FEM_CONN, as well as the special attribute FEM_SPARSE_ELEM.

Without the FEM_SPARSE_ELEM attribute, a sparse element will be copied to every chunk that contains all the sparse element's nodes. This is useful for things like node-associated boundary conditions, where the sparse element connectivity might list the nodes with boundary conditions, and the sparse element data might list the boundary condition values.

The FEM_SPARSE_ELEM attribute lists the ordinary element each sparse element should be partitioned with. This attribute consists of pairs (*elType*,*elNum*), indicating that this sparse element should be sent to wherever the *elNum*'th FEM_ELEM+*elType* is partitioned.

- FEM_SPARSE_ELEM Lists the element we should be partitioned with. The width of this attribute is always 2, and the data type must be an index type-FEM_INDEX_0 or FEM_INDEX_1.

8.3.2 Mesh Entity Manipulation

```
int FEM_Mesh_default_read(void);
```

```
INTEGER function :: FEM_Mesh_default_read()
```

Return the default reading mesh. This routine is valid:

- From driver(), to return the partitioned mesh.
- During your FEM_Update_mesh routine, to return the assembled mesh.
- Anytime after a call to FEM_Mesh_set_default_read.

```
int FEM_Mesh_default_write(void);
```

```
INTEGER function :: FEM_Mesh_default_write()
```

Return the default writing mesh. This routine is valid:

- From init(), to change the new serial mesh.
- From driver(), to change the new partitioned mesh.
- During your FEM_Update_mesh routine, to change the new serial mesh.
- Anytime after a call to FEM_Mesh_set_default_write.

```
int FEM_Mesh_get_length(int mesh,int entity);
```

```
INTEGER function :: FEM_Mesh_get_length(mesh,entity)
```

Return the number of entities that exist in this mesh.

This call can be used with any entity. For example, to get the number of nodes,

```
nNodes=FEM_Mesh_get_length(mesh,FEM_NODE)
```

To get the number of ghost nodes,

```
nGhostNodes=FEM_Mesh_get_length(mesh,FEM_GHOST+FEM_NODE)
```

To get the number of real elements of type 2,

```
nElem=FEM_Mesh_get_length(mesh,FEM_ELEM+2)
```

```
void FEM_Mesh_data(int mesh,int entity,int attr, void *data, int  
first, int length, int datatype,int width);
```

```
SUBROUTINE FEM_Mesh_data(mesh,entity,attr,data,first,length,datatype,width)  
INTEGER, INTENT(IN) :: mesh,entity,attr,first,length,datatype,width  
datatype, intent(inout) :: data(width,length)
```

This is the one routine for getting or setting entity's attributes on the mesh.

- mesh A FEM mesh object. Depending on whether this is a reading or writing mesh, this routine reads from or writes to the data array you pass in.
- entity A FEM entity code, for example FEM_NODE or FEM_GHOST+FEM_ELEM+1.

- attr A FEM attribute code, for example FEM_DATA+*tag* or FEM_CONN.
- data The user data to get or set. Each row of this array consists of width values, and contains the data values of the attribute for the corresponding entity. This data must be formatted as one of:

```
datatype :: data(width,length)
datatype :: data(width*length)
```

- first The first entity to affect. In C, this is normally 0; in Fortran, this is normally 1.
- length The number of entities to affect. The entities affected are thus those numbered from first to first+length-1. For now, length must be either 1, to touch a single entity; or else the total number of entities—that is, FEM_Mesh_get_length(mesh,entity).
- datatype The data type stored in this attribute. This is one of the standard FEM data types FEM_BYTE, FEM_INT, FEM_FLOAT, or FEM_DOUBLE; or else the C-style 0-based index type FEM_INDEX_0 or the Fortran-style 1-based index type FEM_INDEX_1. Alternatively, the equivalent types IDXL_BYTE, IDXL_INT, IDXL_FLOAT, IDXL_DOUBLE, IDXL_INDEX_0, or IDXL_INDEX_1 may be used.
- width The number of data items per entity.

For example, to set the element connectivity, which is stored as 3 integer node indices in nodes, you would:

```
/* C version */
int *nodes=new int [3*nElems];
... fill out nodes ...
FEM_Mesh_data(mesh,FEM_ELEM+1,FEM_CONN, nodes, 0,nElems, FEM_INDEX_0, 3);
... continue to use or delete nodes ...
```

```
! F90 version
ALLOCATE(nodes(3,nElems))
... fill out nodes ...
CALL FEM_Mesh_data(mesh,FEM_ELEM+1,FEM_CONN, nodes, 1,nElems, FEM_INDEX_1, 3)
... continue to use or delete nodes ...
```

To add a new node property with 2 double-precision numbers from an array mat (containing, for example, material properties), you would first pick an unused user data “tag”, for example 13, and:

```
/* C version */
double *mat=new double[2*nNodes];
...
FEM_Mesh_data(mesh,FEM_NODE, FEM_DATA+13, mat, 0,nNodes, FEM_DOUBLE, 2);
```

```
! F90 version
ALLOCATE(mat(2,nNodes))
CALL FEM_Mesh_data(mesh,FEM_NODE,FEM_DATA+13, mat, 1,nNodes, FEM_DOUBLE, 2)
```

8.3.3 Entity Inquiry

```
int FEM_Mesh_get_width(int mesh,int entity,int attr);
```

```
INTEGER function :: FEM_Mesh_get_width(mesh,entity,attr)
INTEGER, INTENT(IN) :: mesh,entity,attr
```

Return the width of the attribute attr of entity of mesh. This is the value previously passed as “width” to FEM_Mesh_data.

```
int FEM_Mesh_get_datatype(int mesh, int entity, int attr);
```

```
INTEGER function :: FEM_Mesh_get_datatype(mesh, entity, attr)
INTEGER, INTENT(IN) :: mesh, entity, attr
```

Return the FEM data type of the attribute attr of entity of mesh. This is the value previously passed as “datatype” to FEM_Mesh_data.

```
int FEM_Mesh_get_entities(int mesh, int *entities);
```

```
INTEGER function :: FEM_Mesh_get_entities(mesh, entities)
INTEGER, INTENT(IN) :: mesh
INTEGER, INTENT(OUT) :: entities(:)
```

Extract an array of the different entities present in this mesh. Returns the number of entity types present. The entities array must be big enough to hold all the different entities in the mesh.

For example, a simple mesh might have two entity types: FEM_NODE and FEM_ELEM+1.

```
int FEM_Mesh_get_attributes(int mesh, int entity, int *attributes);
```

```
INTEGER function :: FEM_Mesh_get_attributes(mesh, entity, attributes)
INTEGER, INTENT(IN) :: mesh, entity
INTEGER, INTENT(OUT) :: attributes(:)
```

Extract an array of the different attributes of this entity. Returns the number of attribute types present. The attributes array must be big enough to hold all the attributes.

For example, a simple element might have three attributes: FEM_CONN for node connectivity, FEM_GLOBALNO for global element numbers, and FEM_DATA+7 for a material type.

```
const char *FEM_Get_entity_name(int entity, char *storage);
const char *FEM_Get_attr_name(int attr, char *storage);
const char *FEM_Get_datatype_name(int datatype, char *storage);
```

Return a human-readable name for this FEM entity, attribute, or datatype. The storage array must point to a buffer of at least 100 characters; this array might be used as temporary space to store the returned string.

These routines are only available in C.

8.3.4 Advanced Entity Manipulation

```
void FEM_Mesh_data_offset(int mesh, int entity, int attr, void *data,
int first, int length, int datatype, int width, int offsetBytes, int
distanceBytes, int skewBytes);
```

```
SUBROUTINE FEM_Mesh_data_offset(mesh, entity, attr, data, first, length, datatype, width,
offsetBytes, distanceBytes, skewBytes)
INTEGER, INTENT(IN) :: mesh, entity, attr, first, length, datatype, width
INTEGER, INTENT(IN) :: offsetBytes, distanceBytes, skewBytes
datatype, intent(inout) :: data(width, length)
```

This routine is a more complicated version of FEM_Mesh_data. It allows you to get or set a mesh field directly from a user-defined structure. See the documentation of IDXL_Layout_offset in Section 8.8.2 for details on how to set offsetBytes, distanceBytes, and skewBytes.


```
void FEM_Mesh_data_layout(int mesh, int entity, int attr, void *data,
int firstItem, int length, IDXL_Layout_t layout);
```

```
SUBROUTINE FEM_Mesh_data_layout(mesh, entity, attr, data, first, length, layout)
INTEGER, INTENT(IN) :: mesh, entity, attr, first, length, layout
INTEGER, INTENT(IN) :: layout
```

This routine is a more complicated version of FEM_Mesh_data. Like FEM_Mesh_data_offset, it allows you to get or set a mesh field directly from a user-defined structure; but this routine expects the structure to be described by an IDXL_Layout object.

8.4 Meshes

A “mesh” is a collection of nodes and elements knit together in memory, as described in Section 8.1.2. Meshes are always referred to by an integer that serves as a handle to the local mesh.

This section describes routines to manipulate entire meshes at once: this includes calls to create and delete meshes, read and write meshes, partition and reassemble meshes, and send meshes between processors.

Only a few of the mesh routines are collective; most of them only describe local data and hence operate independently on each chunk.

8.4.1 Mesh Routines

```
int FEM_Mesh_allocate(void);
```

```
INTEGER FUNCTION :: FEM_Mesh_allocate()
```

Create a new local mesh object. The mesh is initially empty, but it is a setting mesh, so call FEM_Mesh_data to fill the mesh with data.

```
int FEM_Mesh_deallocate(int mesh);
```

```
SUBROUTINE FEM_Mesh_deallocate(mesh)
INTEGER, INTENT(IN) :: mesh
```

Destroy this local mesh object, and its associated data.

```
int FEM_Mesh_copy(int mesh);
```

```
INTEGER FUNCTION FEM_Mesh_copy(mesh)
INTEGER, INTENT(IN) :: mesh
```

Create a new mesh object with a separate copy of the data stored in this old mesh object.

```
void FEM_Mesh_write(int mesh, const char *prefix, int partNo, int
nParts);
```

```
SUBROUTINE FEM_Mesh_write(mesh, prefix, partNo, nParts)
INTEGER, INTENT(IN) :: mesh
INTEGER, INTENT(IN) :: partNo, nParts
character (LEN=*) , INTENT(IN) :: prefix
```

Write this mesh to the file “prefix_vppartNo_nParts.dat”.

By convention, partNo begins at 0; but no index conversion is performed so you can assign any meaning to partNo and nParts. In particular, this routine is not collective—you can read any mesh from any processor. For example, if prefix is “foo/bar”, the data for the first of 7 chunks would be stored in “foo/bar_vp0_7.dat” and could be read using FEM_Mesh_read(‘foo/bar’,0,7).

Meshes are stored in a machine-portable format internal to FEM. The format is currently ASCII based, but it is subject to change. We strongly recommend using the FEM routines to read and write these files rather than trying to prepare or parse them yourself.

```
int FEM_Mesh_read(const char *prefix,int partNo,int nParts);
```

```
INTEGER FUNCTION :: FEM_Mesh_read(prefix,partNo,nParts)
INTEGER, INTENT(IN) :: partNo,nParts
character (LEN=*) , INTENT(IN) :: prefix
```

Read a new mesh from the file “prefix_vppartNo_nParts.dat”. The new mesh begins in getting mode, so you can read the data out of the mesh using calls to FEM_Mesh_data.

```
int FEM_Mesh_broadcast(int mesh,int fromRank,FEM_Comm_t comm_context);
```

```
INTEGER FUNCTION :: FEM_Mesh_broadcast(mesh,fromRank,comm_context)
INTEGER, INTENT(IN) :: mesh,fromRank,comm_context
```

Take the mesh mesh on processor fromRank (normally 0), partition the mesh into one piece per processor (in the MPI communicator comm_context, and return each processor its own piece of the partitioned mesh. This call is collective, but only processor fromRank needs to pass in a mesh; the mesh value is ignored on other processors.

For example, if rank 0 has a mesh named “src”, we can partition src for all the processors by executing:

```
m=FEM_Mesh_broadcast(src,0,MPI_COMM_WORLD);
```

The new, partitioned mesh is in getting mode, so you can read the partitioned data using calls to FEM_Mesh_data. This call does not affect mesh in any way.

```
int FEM_Mesh_reduce(int mesh,int toRank,FEM_Comm_t comm_context);
```

```
INTEGER FUNCTION :: FEM_Mesh_reduce(mesh,toRank,comm_context)
INTEGER, INTENT(IN) :: mesh,toRank,comm_context
```

This call is the reverse operation of FEM_Mesh_broadcast: each processor passes in a mesh in mesh, the mesh is assembled, and the function returns the assembled mesh to processor toRank. This call is collective, but only processor toRank is returned a mesh; all other processors are returned the non-mesh value 0.

The new, reassembled mesh is in getting mode. This call does not affect mesh.

8.4.2 Mesh Utility

```
int FEM_Mesh_is_get(int mesh)
```

```
INTEGER FUNCTION :: FEM_Mesh_is_get(mesh)
INTEGER, INTENT(IN) :: mesh
```

Return true if this mesh is in getting mode. A getting mesh returns values to FEM_Mesh_data.

```
int FEM_Mesh_is_set(int mesh)
```

```
INTEGER FUNCTION :: FEM_Mesh_is_set(mesh)
INTEGER, INTENT(IN) :: mesh
```

Return true if this mesh is in setting mode. A setting mesh extracts values from FEM_Mesh_data.

```
void FEM_Mesh_become_get(int mesh)
```

```
SUBROUTINE :: FEM_Mesh_become_get(mesh)
INTEGER, INTENT(IN) :: mesh
```

Put this mesh in getting mode, so you can read back its values.

```
void FEM_Mesh_become_set(int mesh)
```

```
SUBROUTINE :: FEM_Mesh_become_set(mesh)
INTEGER, INTENT(IN) :: mesh
```

Put this mesh in setting mode, so you can set its values.

```
void FEM_Mesh_print(int mesh);
```

```
SUBROUTINE FEM_Mesh_print(mesh)
INTEGER, INTENT(IN) :: mesh
```

Print out a text description of the nodes and elements of this mesh.

8.4.3 Advanced Mesh Manipulation

```
typedef void (*FEM_Userdata_fn)(pup_er p, void *data);
void FEM_Mesh_pup(int mesh, int pupTag, FEM_Userdata_fn fn, void *data);
```

```
SUBROUTINE myPupFn(p, data)
INTEGER, INTENT(IN) :: p
TYPE(myType) :: data

SUBROUTINE FEM_Mesh_pup(mesh, pupTag, myPupFn, data)
INTEGER, INTENT(IN) :: mesh, pupTag
SUBROUTINE :: myPupFn
TYPE(myType) :: data
```

Store data with this mesh. data is a struct or TYPE with a pup function myPupFn—see the TCharm manual for details on writing a pup function. pupTag is an integer used to distinguish different pieces of data associated with this mesh.

When called on a setting mesh, this routine stores data; when called on a getting mesh, this routine reads out data.

data will be associated with the mesh itself, not any entity in the mesh. This makes it useful for storing shared data, often simulation constants such as the timestep or material properties. data is made a part of the mesh, and it will be read and written, sent and received, partitioned and assembled with the mesh.

```
void FEM_Mesh_send(int mesh, int toRank, int tag, FEM_Comm_t
comm_context);
```

```
SUBROUTINE FEM_Mesh_send(mesh,toRank,tag,comm)
INTEGER, INTENT(IN) :: mesh,toRank,tag,comm
```

Send the mesh mesh to the processor toRank, using the MPI tag tag and communicator comm_context. Tags are normally only needed if you plan to mix direct MPI calls with your FEM calls.

This call does not affect mesh.

```
int FEM_Mesh_recv(int fromRank,int tag,FEM_Comm_t comm_context);
```

```
INTEGER FUNCTION FEM_Mesh_recv(fromRank,tag,comm)
INTEGER, INTENT(IN) :: fromRank,tag,comm
```

Receive a new mesh from the processor fromRank, using the MPI tag tag and communicator comm_context. You can also use the special values MPI_ANY_SOURCE as fromRank to receive a mesh from any processor, or use MPI_ANY_TAG for tag to match any tag.

The new mesh is returned in getting mode.

```
void FEM_Mesh_partition(int mesh,int nParts,int *destMeshes);
```

```
SUBROUTINE FEM_Mesh_partition(mesh,nParts,destMeshes)
INTEGER, INTENT(IN) :: mesh,nParts
INTEGER, INTENT(OUT) :: destMeshes(nParts)
```

Divide mesh into nParts pieces, and store the pieces into the array destMeshes.

The partitioned mesh is returned in getting mode. This is a local call; FEM_Mesh_broadcast is the collective version. This call does not affect the source mesh mesh.

```
int FEM_Mesh_assemble(int nParts,const int *srcMeshes);
```

```
INTEGER FUNCTION FEM_Mesh_assemble(nParts,srcMeshes)
INTEGER, INTENT(IN) :: nParts, srcMeshes(nParts)
```

Assemble the nParts meshes listed in srcMeshes into a single mesh. Corresponding mesh pieces are matched using the attribute FEM_GLOBALNO. Specifically, if the value of the integer index attribute FEM_GLOBALNO for an entity is i , the entity will be given the number i in the reassembled mesh. If you do not set FEM_GLOBALNO, the different pieces of the mesh will remain separate—even “matching” nodes will not be merged.

The assembled mesh is returned in getting mode. This is a local call; FEM_Mesh_reduce is the collective version. This call does not affect the source meshes.

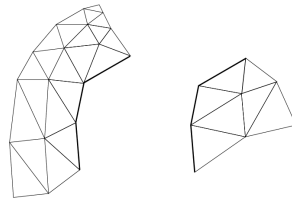
```
void FEM_Mesh_copy_globalno(int src_mesh,int dest_mesh);
```

```
SUBROUTINE FEM_Mesh_copy_globalno(src_mesh,dest_mesh)
INTEGER, INTENT(IN) :: src_mesh,dest_mesh
```

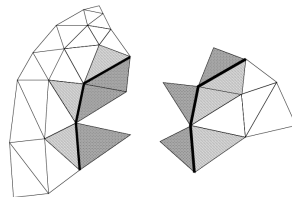
Copy the FEM_GLOBALNO attribute for all the entity types in src_mesh into all the matching types in dest_mesh, where the matching types exist. This call is often used before an FEM_Mesh_assemble or FEM_Mesh_reduce to synchronize global numbers before reassembly.

8.5 Mesh Ghosts

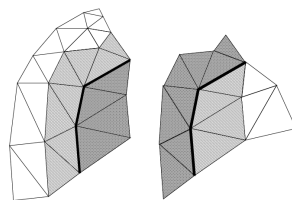
A **ghost entity** is a local, read-only copy of a real entity on another chunk. Ghosts are typically added to the boundary of a chunk to allow the real (non-ghost) elements at the boundary to access values across the processor boundary. This makes a chunk “feel” as if it was part of a complete unpartitioned mesh; and can be useful with cell-centered methods, and in mesh modification.



9: A small mesh partitioned into two pieces.



10: The same mesh with one layer of edge-adjacent ghosts.



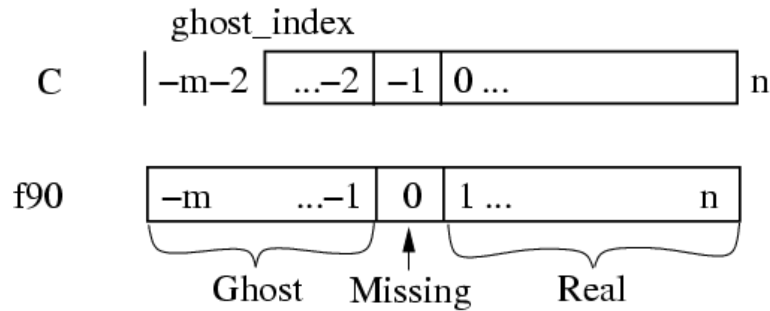
11: The same mesh with one layer of node-adjacent ghosts.

In Figure 9, we begin with a small mesh partitioned into pieces on the left and right. In Figure 10, we have added ghost elements (dark hashing) that share an edge with adjacent real elements (light hatching). In Figure 11, we add ghost elements that share at least one node with adjacent real elements.

8.5.1 Ghost Numbering

Ghosts and real entities are stored by the framework in separate lists—to access the ghost entity type, add FEM_GHOST to the real entity’s type. For example, FEM_GHOST+FEM_ELEM+1 lists the ghost elements for elType 1. To get the number of ghost nodes, you would call FEM_Mesh_get_length(mesh,FEM_GHOST+FEM_NODE).

For real elements, the element connectivity always consists of real nodes. But for ghost elements, the adjacent nodes may be missing, or may themselves be ghosts. Thus ghost element connectivity lists may include the invalid value -1 (in C) or 0 (in Fortran) to indicate that the corresponding node is not present; or may include values less than this, which indicate the corresponding node is a ghost. In C, ghost node i is indicated by the value $-2 - i$, while in Fortran, ghost node i is indicated by the value $-i$. This node indexing system is illustrated in Figure 12. This indexing system is bizarre, but it allows us to keep the real and ghost nodes clearly separate, while still allowing real and ghost nodes to be added in increasing order at both ends.



12: Node indices used in the element connectivity array. There are n real nodes and m ghosts.

Since the C tests are complicated, in C we recommend using these macros:

- `FEM_Is_ghost_index(i)` returns true if i represents a ghost node. In Fortran, use the test $i \leq 0$.
- `FEM_From_ghost_index(i)` returns the ghost node's index given its connectivity entry. In Fortran, use the expression $-i$.
- `FEM_To_ghost_index(i)` returns the connectivity entry for a given ghost node index. In Fortran, again use the expression $-i$.

For example, a quadrilateral ghost element that is adjacent to, respectively, two real nodes 23 and 17, the tenth local ghost node, and one not-present node might have a connectivity entry of 23,17,-11,-1 (in C) or 23,17,-10,0 (in Fortran).

Applications may wish to use some other numbering, such as by storing all the ghost nodes after all the real nodes. The code to extract and renumber the connectivity of some 3-node triangles stored in `FEM_ELEM+2` would be:

```
/* C version */
int nReal=FEM_Mesh_get_length(mesh,FEM_ELEM+2);
int nGhost=FEM_Mesh_get_length(mesh,FEM_GHOST+FEM_ELEM+2);
typedef int intTriplet[3];
intTriplet *conn=new intTriplet[nReal+nGhost];
/* Extract real triangles into conn[0..nReal-1] */
FEM_Mesh_data(mesh,FEM_ELEM+2,FEM_CONN, &conn[0][0], 0,nReal, 3,FEM_INDEX_0);
/* Extract ghost triangles into conn[nReal..nReal+nGhost-1] */
FEM_Mesh_data(mesh,FEM_GHOST+FEM_ELEM+2,FEM_CONN, &conn[nReal][0], 0,nGhost, 3,FEM_
↪INDEX_0);

/* Renumber the ghost triangle connectivity */
for (int t=nReal;t<nReal+nGhost;t++)
  for (int i=0;i<3;i++) {
    int in=conn[t][i]; /* uses FEM ghost node numbering */
    int out; /* uses application's ghost numbering */
    if (in==-1) {
      out=some_value_for_missing_nodes;
    } else if (FEM_Is_ghost_index(in)) {
      out=first_application_ghost+FEM_From_ghost_index(in);
    } else /*regular real node*/ {
      out=in;
    }
    conn[t][i]=out;
  }
}
```

```
! F90 version
INTEGER, ALLOCATABLE :: conn(3,:)
INTEGER :: nReal,nGhost,t,i,in,out
```

(continues on next page)

(continued from previous page)

```

nReal=FEM_Mesh_get_length(mesh,FEM_ELEM+2)
nGhost=FEM_Mesh_get_length(mesh,FEM_GHOST+FEM_ELEM+2)
ALLOCATE (conn(3,nReal+nGhost))
! Extract real triangles into conn[1..nReal]
CALL FEM_Mesh_data(mesh,FEM_ELEM+2,FEM_CONN, conn, 1,nReal, 3,FEM_INDEX_1)
! Extract ghost triangles into conn[nReal+1..nReal+nGhost]
CALL FEM_Mesh_data(mesh,FEM_GHOST+FEM_ELEM+2,FEM_CONN, conn(1,nReal+1), 1,nGhost, 3,
↳FEM_INDEX_1)

! Renumber the ghost triangle connectivity
DO t=nReal+1,nReal+nGhost
  DO i=1,3
    in=conn(i,t)
    IF (in .EQ. 0) out=some_value_for_missing_nodes
    IF (in .LT. 0) out=first_application_ghost-1+(-in)
    IF (in .GT. 0) out=in
    conn(i,t)=out
  END DO
END DO

```

8.5.2 Setting up the ghost layer

The framework's ghost handling is element-centric. You specify which kinds of elements should be ghosts and how they connect by listing their faces before partitioning.

```
void FEM_Add_ghost_layer(int nodesPerFace,int doAddNodes);
```

```
SUBROUTINE FEM_Add_ghost_layer(nodesPerFace,doAddNodes)
INTEGER, INTENT(IN) :: nodesPerFace,doAddNodes
```

This routine creates a new layer of ghosts around each FEM chunk. `nodesPerFace` is the number of shared nodes that together form a “face”. `doAddNodes` specifies that you want ghost nodes around your ghost elements. If `doAddNodes` is 0, ghost elements will have invalid -1 (in C) or 0 (in Fortran) connectivity entries where there is no corresponding local node.

A face is an unordered “tuple” of nodes, and is an abstract way to describe which ghosts your application needs—an element will be added to your chunk if it connects to at least one of your elements' faces. For example, if you have a 3D, tetrahedral element that require ghosts on all 4 of its sides, this is equivalent to requiring ghosts of every element that shares all 3 nodes of one of your triangular faces, so for you a face is a 3-node triangle. If you have a 2D shape and want edge-adjacency, for you a face is a 2-node edge. If you want node-adjacent ghosts, a face is a single node.

Calling this routine several times creates several layers of ghost elements, and the different layers need not have the same parameters.

```
void FEM_Add_ghost_elem(int elType,int facesPerElem,const int
*elem2face);
```

```
SUBROUTINE FEM_Add_ghost_elem(elType,facesPerElem,elem2face)
INTEGER, INTENT(IN) :: elType,facesPerElem
INTEGER, INTENT(IN) :: elem2face(nodesPerFace,facesPerElem)
```

This call is used to specify which type of element is to be added to the current ghost layer. `facesPerElem` and `elem2face` specify a mapping between each element and the surrounding faces. The `elem2face` table lists, for each face, the nodes of this element which form the face, specified as element-local numbers—indices into this element's connectivity

entry. The `elem2face` table should have `nodesPerFace*facesPerElem` entries, and no entry should be greater than `nodePerEl` for that element type.

Because all faces must take up the same space in the array, `elem2face` can include special indices— -1 for C, 0 for Fortran—that indicate the corresponding face is actually shorter than usual. For example, if `nodesPerFace` for this layer is 4, for 4-node quadrilateral faces, you could set one entry in `elem2face` to -1 to specify this is a 3-node triangular face. Faces of different lengths will never match, so this is just a simple way to add ghosts from two kinds of faces at once.

The above two routines are always used together. For example, if your elements are 3-node triangles and you only require one shared node for inclusion in a single ghost layer, you would use:

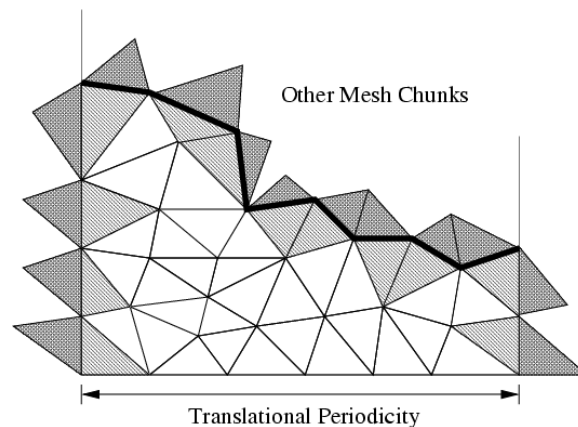
```
FEM_Add_ghost_layer(1,1); /* 1 node per face: node adjacency */
const static int tri2node[]={0,1,2};
FEM_Add_ghost_elem(0,3,tri2node); /* triangles are surrounded by 3 nodes */
```

If you require two shared nodes (a shared edge), the code will look like:

```
FEM_Add_ghost_layer(2,1); /* 2 nodes per face: edge adjacency */
const static int tri2edge[]={0,1, 1,2, 2,0};
FEM_Add_ghost_elem(0,3,tri2edge); /*triangles are surrounded by 3 edges */
```

8.5.3 Symmetries and Ghosts-Geometric Layer

The FEM framework can create ghosts not only of things that are on other processors, but also for various problem symmetries, like mirror reflection, and various types of periodicities. The interface for these ghosts is simple—you ask for the symmetries to be created, then you will get extra ghosts along each symmetry boundary. The symmetry ghosts are updated properly during any communication, even if the symmetry ghosts are ghosts of real local elements from the same chunk.



13: Illustrating symmetry ghost elements.

Figure 13 shows a chunk of a mesh for a rectangular domain with horizontal linear translational periodicity—that is, the domain repeats horizontally. Symmetry ghosts lie along the left and right sides; ordinary cross-processor parallel ghosts lie along the top edge where this chunk joins up with the rest of the domain; and the external boundary along the bottom of the chunk has no ghosts.

```
void FEM_Add_linear_periodicity( int nFaces, int nPer, const int
*facesA, const int *facesB, int nNodes, const double *nodeLocs );
```



```

SUBROUTINE FEM_Add_linear_periodicity(nFaces,nPer,facesA,facesB,
nNodes,nodeLocs)
INTEGER, INTENT(IN) :: nFaces, nPer, nNodes
INTEGER, INTENT(IN) :: facesA(nPer,nFaces), facesB(nPer,nFaces)
double precision, intent(IN) :: nodeLocs(3,nNodes)

```

Make facesA and facesB match up under linear translation. Each face of facesA must match up with exactly one face of facesB, but both the faces and the nodes within a face can be permuted in any order—the order is recovered by matching 3d locations in the nodeLocs array.

This call can be repeated, for example if the domain is periodic along several directions. This call can only be issued from init().

```

void FEM_Sym_coordinates(int elTypeOrMinusOne,double *locs);

```

```

SUBROUTINE FEM_Sym_coordinates(elTypeOrZero,locs)
INTEGER, INTENT(IN) :: elTypeOrZero
double precision, intent(inout) :: locs(3,<number of items>)

```

This call adjusts the 3d locations listed in locs so they respect the symmetries of their corresponding item. If elTypeOrZero is an element type, the locations are adjusted to match with the corresponding element; if elTypeOrZero is zero, the locations are adjusted to match up with the corresponding node.

This call is needed because symmetry ghost nodes and elements initially have their original locations, which must be adjusted to respect the symmetry boundaries. Thus this call is needed both for initial location data (e.g., from FEM_Get_node_data) as well as any communicated location data (e.g., from FEM_Update_ghost_field).

This call can only be issued from driver().

8.5.4 Advanced Symmetries and Ghosts-Lower Layer

The geometric symmetry layer in the preceding section is actually a thin wrapper around this lower, more difficult to use layer.

```

void FEM_Set_sym_nodes(const int *canon,const int *sym);

```

```

SUBROUTINE FEM_Set_sym_nodes(canon,sym)
INTEGER, INTENT(IN) :: canon(nNodes)
INTEGER, INTENT(IN) :: sym(nNodes)

```

This call describes all possible symmetries in an extremely terse format. It can only be called from init(). The “canonicalization array” canon maps nodes to their canonical representative—if $\text{canon}(i)=\text{canon}(j)$, nodes i and j are images of each other under some symmetry. The sym array has bits set for each symmetry boundary passing through a node.

For example, a 2d domain with 6 elements A, B, C, D, E, and F and 12 nodes numbered 1-12 that is mirror-symmetric on the horizontal boundaries but periodic in the vertical boundaries would look like:

D [^] '		D [^]		E [^]		F [^]		F ^{^`}
- 1	-	2	-	3	-	4	-	
A'		A		B		C		C [`]
- 5	-	6	-	7	-	8	-	
D'		D		E		F		F [`]
- 9	-	10	-	11	-	12	-	
Av'		Av		Bv		Cv		Cv [`]

(continues on next page)

(continued from previous page)

```
v indicates the value has been shifted down (bottom boundary),
^ indicates the value has been shifted up (top boundary),
' indicates the value has been copied from the left (right boundary),
` indicates the value has been copied from the right (left boundary).
```

If we mark the left border with 1, the top with 2, the right with 4, and the bottom with 8, this situation is indicated by topologically pasting the top row to the bottom row by setting their canon entries equal, and marking each node with its symmetries.

Node	canon	sym
1	1	3 (left + top)
2	2	2 (top)
3	3	2 (top)
4	4	6 (top + right)
5	5	1 (left)
6	6	0 (none)
7	7	0 (none)
8	8	4 (right)
9	1	9 (left+bottom)
10	2	8 (bottom)
11	3	8 (bottom)
12	4	12 (bottom+right)

```
void FEM_Get_sym(int elTypeOrMinusOne, int *destSym);
```

```
SUBROUTINE FEM_Get_sym(elTypeOrZero, destSym);
INTEGER, INTENT (IN) :: elTypeOrMinusOne
INTEGER, INTENT (OUT) :: destSym(nItems)
```

This call extracts the list of symmetry conditions that apply to an item type. If `elType` is an element type, it returns the symmetry conditions that apply to that element type; if `elType` is -1 (zero for Fortran), it returns the symmetry conditions that apply to the nodes. Symmetry conditions are normally only nonzero for ghost nodes and elements.

Mirror symmetry conditions are not yet supported, nor are multiple layers of symmetry ghosts, but both should be easy to add without changing this interface.

8.6 Older Mesh Routines

These routines have a simpler, but less flexible interface than the general routines described in Section 8.3. Because they are easy to implement in terms of the new routines, they will remain part of the framework indefinitely. These routines always use the default mesh, as returned by `FEM_Mesh_default_read` and `FEM_Mesh_default_write`.

```
void FEM_Set_elem(int elType, int nEl, int doublePerEl, int nodePerEl);
void FEM_Get_elem(int elType, int *nEl, int *doublePerEl, int
*nodePerEl);
```

```
SUBROUTINE FEM_Set_elem(elType, nEl, doublePerEl, nodePerEl)
INTEGER, INTENT (IN) :: elType, nEl, doublePerEl, nodePerEl
SUBROUTINE FEM_Get_elem(elType, nEl, doublePerEl, nodePerEl)
```

(continues on next page)

(continued from previous page)

```

INTEGER, INTENT(IN) :: elType
INTEGER, INTENT(OUT) :: nEl, doublePerEl, nodePerEl

```

Describe/retrieve the number and type of elements. `elType` is a user-defined small, unique element type tag. `nEl` is the number of elements being registered. `doublePerEl` and `nodePerEl` are the number of doubles of user data, and nodes (respectively) associated with each element.

`doublePerEl` or `nodePerEl` may be zero, indicating that no user data or connectivity data (respectively) is associated with the element.

You can make this and any other mesh setup calls in any order—there is no need to make them in linearly increasing order. However, for a given type of element `FEM_Set_elem` must be called before setting that element's connectivity or data.

```

void FEM_Set_elem_conn(int elType, const int *conn);
void FEM_Get_elem_conn(int elType, int *conn);

```

```

SUBROUTINE FEM_Set_elem_conn_r(elType, conn)
INTEGER, INTENT(IN) :: elType
INTEGER, INTENT(IN), dimension(nodePerEl, nEl) :: conn
SUBROUTINE FEM_Get_elem_conn_r(elType, conn)
INTEGER, INTENT(IN) :: elType
INTEGER, INTENT(OUT), dimension(nodePerEl, nEl) :: conn
SUBROUTINE FEM_Set_elem_conn_c(elType, conn)
INTEGER, INTENT(IN) :: elType
INTEGER, INTENT(IN), dimension(nEl, nodePerEl) :: conn
SUBROUTINE FEM_Get_elem_conn_c(elType, conn)
INTEGER, INTENT(IN) :: elType
INTEGER, INTENT(OUT), dimension(nEl, nodePerEl) :: conn

```

Describe/retrieve the element connectivity array for this element type. The connectivity array is indexed by the element number, and gives the indices of the nodes surrounding the element. It is hence `nodePerEl*nEl` integers long.

The C version array indices are zero-based, and must be stored in row-major order (a given element's surrounding nodes are stored contiguously in the `conn` array). The Fortran version indices are one-based, and are available in row-major (named `_r`) and column-major (named `_c`) versions. We recommend row-major storage because it results in better cache utilization (because the nodes around an element are stored contiguously).

In this older interface, ghost nodes are indicated by invalid,

```

void FEM_Set_node(int nNode, int doublePerNode);
void FEM_Get_node(int
    *nNode, int *doublePerNode);

```

```

SUBROUTINE FEM_Set_node(nNode, doublePerNode)
INTEGER, INTENT(IN) :: nNode, doublePerNode
SUBROUTINE FEM_Get_node(nNode, doublePerNode)
INTEGER, INTENT(OUT) :: nNode, doublePerNode

```

Describe/retrieve the number of nodes and doubles of user data associated with each node. There is only one type of node, so no `nodeType` identifier is needed.

`doublePerNode` may be zero, indicating that no user data is associated with each node.

8.6.1 Old Mesh Data

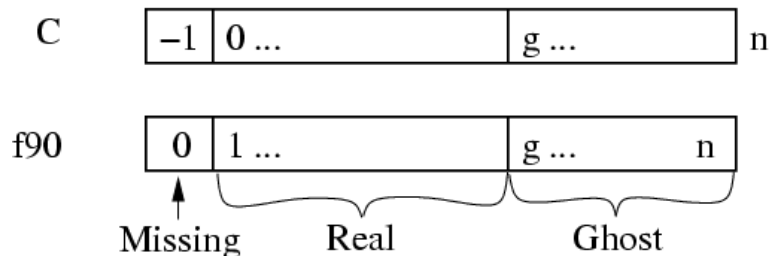
```
void FEM_Set_node_data(const double *data);
void FEM_Get_node_data(double *data);
void FEM_Set_elem_data(int elType, const double *data);
void FEM_Get_elem_data(int elType, double *data);
```

```
SUBROUTINE FEM_Set_node_data_r(data)
REAL*8, INTENT(IN), dimension(doublePerNode,nNode) :: data
SUBROUTINE FEM_Get_node_data_r(data)
REAL*8, INTENT(OUT), dimension(doublePerNode,nNode) :: data
SUBROUTINE FEM_Set_node_data_c(data)
REAL*8, INTENT(IN), dimension(nNode,doublePerNode) :: data
SUBROUTINE FEM_Get_node_data_c(data)
REAL*8, INTENT(OUT), dimension(nNode,doublePerNode) :: data
SUBROUTINE FEM_Set_elem_data_r(elType,data)
INTEGER, INTENT(IN) :: elType
REAL*8, INTENT(IN), dimension(doublePerElem,nElem) :: data
SUBROUTINE FEM_Get_elem_data_r(elType,data)
INTEGER, INTENT(IN) :: elType
REAL*8, INTENT(OUT), dimension(doublePerElem,nElem) :: data
SUBROUTINE FEM_Set_elem_data_c(elType,data)
INTEGER, INTENT(IN) :: elType
REAL*8, INTENT(IN), dimension(nElem,doublePerElem) :: data
SUBROUTINE FEM_Get_elem_data_c(elType,data)
INTEGER, INTENT(IN) :: elType
REAL*8, INTENT(OUT), dimension(nElem,doublePerElem) :: data
```

Describe/retrieve the optional, uninterpreted user data associated with each node and element. This user data is partitioned and reassembled along with the connectivity matrix, and may include initial conditions, node locations, material types, or any other data needed or produced by the program. The Fortran arrays can be row- or column- major (see FEM_Set_elem_conn for details). The row-major form is preferred.

8.6.2 Old Ghost Numbering

In this older version of the framework, FEM_Get_node and FEM_Get_elem return the **total** number of nodes and elements, including ghosts. The routines below return the index of the first ghost node or element, where ghosts are numbered after all the real elements. This old ghost numbering scheme does not work well when adding new ghosts, which is why the new ghost numbering scheme describes in Section 8.5.1 is used in the new API.



14: Old ghost element and node numbering. FEM_Get_ghost_returns g , FEM_Get_returns n .

```
int FEM_Get_node_ghost(void);
int FEM_Get_elem_ghost(int elemType);
```

The examples below iterate over the real and ghost elements using the old numbering:

```
// C version:
int firstGhost,max;
FEM_Get_node(&max, &ignored);
firstGhost=FEM_Get_node_ghost();
for (i=0;i<firstGhost;i++)
    //... i is a real node...
for (i=firstGhost;i<max;i++)
    //... i is a ghost node ...
```

```
! Fortran version:
call FEM_Get_node(max,ignored);
firstGhost=FEM_Get_node_ghost();
do i=1,firstGhost-1
!     ... i is a real node...
end do
do i=firstGhost,max
!     ... i is a ghost node...
end do
```

8.6.3 Old Backward Compatibility

```
void FEM_Set_mesh(int nElem, int nNodes, int nodePerEl, const int*
conn);
```

This is a convenience routine equivalent to:

```
FEM_Set_node(nNodes, 0);
FEM_Set_elem(0,nElem,0,nodePerEl);
FEM_Set_elem_Conn(0, conn);
```

```
SUBROUTINE FEM_Set_mesh(nElem,nNodes,nodePerEl, conn)
```

This is a convenience routine equivalent to:

```
CALL FEM_Set_node(nNodes, 0)
CALL FEM_Set_elem(1,nElem,0,nodePerEl)
CALL FEM_Set_elem_Conn_c(1, conn)
```

8.6.4 Old Sparse Data

Sparse data is typically used to represent boundary conditions. For example, in a structural dynamics program typically some nodes have an imposed force or position. The routines in this section are used to describe this kind of mesh-associated data—data that only applies to some “sparse” subset of the nodes or elements.

```
void FEM_Set_sparse(int S_id,int nRec, const int *nodes,int
nodesPerRec, const void *data,int dataPerRec,int dataType);
```

```
SUBROUTINE FEM_Set_sparse(S_id,nRec,nodes,nodesPerRec,data,dataPerRec,dataType)
INTEGER, INTENT(IN) :: S_id,nRec,nodesPerRec,dataPerRec,dataType
INTEGER, INTENT(IN) :: nodes(nodesPerRec,nRec)
varies, INTENT(IN) :: data(dataPerRec,nRec)
```

Register `nRec` sparse data records with the framework under the number `S_id`. The first call to `FEM_Set_sparse` must give a `S_id` of zero in C (1 in fortran); and subsequent calls to `FEM_Set_sparse` must give increasing consecutive `S_ids`.

One sparse data record consists of some number of nodes, listed in the `nodes` array, and some amount of user data, listed in the `data` array. Sparse data records are copied into the chunks that contains all that record's listed nodes. Sparse data records are normally used to describe mesh boundary conditions- for node-associated boundary conditions, `nodesPerRec` is 1; for triangle-associated boundary conditions, `nodesPerRec` is 3.

In general, `nodePerRec` gives the number of nodes associated with each sparse data record, and `nodes` gives the actual node numbers. `dataPerRec` gives the number of data items associated with each sparse data record, and `dataType`, one of `FEM_BYTE`, `FEM_INT`, `FEM_REAL`, or `FEM_DOUBLE`, gives the type of each data item. As usual, you may change or delete the nodes and data arrays after this call returns.

For example, if the first set of sparse data is 17 sparse data records, each containing 2 nodes stored in `bNodes` and 3 integers stored in `bDesc`, we would make the call:

```
/*C version*/
FEM_Set_sparse(0,17, bNodes,2, bDesc,3,FEM_INT);
```

```
! Fortran version
CALL FEM_Set_sparse(1,17, bNodes,2, bDesc,3,FEM_INT)
```

```
void FEM_Set_sparse_elem(int S_id,const int *rec2elem);
```

```
SUBROUTINE FEM_Set_sparse_elem(S_id,rec2elem)
  INTEGER, INTENT(IN) :: S_id
  INTEGER, INTENT(IN) :: rec2elem(2,nRec)
```

Attach the previously-set sparse records `S_id` to the given elements. `rec2elem` consists of pairs of integers—one for each sparse data record. The first integer in the pair is the element type to attach the sparse record to, and the second integer gives the element number within that type. For example, to attach the 3 sparse records at `S_id` to the elements numbered 10, 11, and 12 of the element type `elType`, use:

```
/*C version*/
int rec2elem[]={elType,10, elType,11, elType,12};
FEM_Set_sparse_elem(S_id,rec2elem);
```

```
! Fortran version
integer :: rec2elem(2,3);
rec2elem(1,:)=elType
rec2elem(2,1)=10; rec2elem(2,2)=11; rec2elem(2,3)=12;
CALL FEM_Set_sparse_elem(S_id,rec2elem)
```

```
int FEM_Get_sparse_length(int S_id);
void FEM_Get_sparse(int S_id,int *nodes,void *data);
```

```
function FEM_Get_sparse_length(S_id);
  INTEGER, INTENT(IN) :: S_id
  INTEGER, INTENT(OUT) :: FEM_Get_sparse_Length
  SUBROUTINE FEM_Get_sparse(S_id,nodes,data);
  INTEGER, INTENT(IN) :: S_id
  INTEGER, INTENT(OUT) :: nodes(nodesPerRec,FEM_Get_sparse_Length(S_id))
  varies, INTENT(OUT) :: data(dataPerRec,FEM_Get_sparse_Length(S_id))
```

Retrieve the previously registered sparse data from the framework. `FEM_Get_sparse_length` returns the number of

records of sparse data registered under the given S_id; zero indicates no records are available. FEM_Get_sparse returns you the actual nodes (translated to local node numbers) and unchanged user data for these sparse records.

In this old interface, there is no way to access sparse ghosts.

8.7 Mesh Modification

```
void FEM_Update_mesh(FEM_Update_mesh_fn routine, int
callMeshUpdated, int doWhat);
```

```
SUBROUTINE FEM_Update_mesh(routine, callMeshUpdated, doWhat)
external, INTENT(IN) :: routine
INTEGER, INTENT(IN) :: callMeshUpdated, doWhat
```

Reassemble the mesh chunks from each partition into a single serial mesh, and call the given routine on the assembled mesh. In this routine, which runs on processor 0, the FEM_Get and FEM_Set routines can manipulate the serial mesh. The parameter callMeshUpdated, which must be non-zero, is passed down to routine as routine(callMeshUpdated).

FEM_Get calls from driver() will only return the new mesh after a FEM_Update_mesh call where doWhat is FEM_MESH_UPDATE; otherwise FEM_Get from driver() will still return the old mesh. FEM_Update_mesh can only be called from driver; and must be called by the driver routine for every chunk.

doWhat	Numeric	Repartition?	FEM_Update_mesh
FEM_MESH_OUTPUT	0	No	driver() continues alongside routine
FEM_MESH_FINALIZE	2	No	driver() blocks until routine finishes
FEM_MESH_UPDATE	1	Yes	driver() blocks for the new partition

For example, FEM_Update_mesh(my_output_routine, k, FEM_MESH_OUTPUT) reassembles the mesh and calls a routine named my_output_routine(k) while the driver routines continue with the computation. This might be useful, for example, for writing out intermediate solutions as a single file; writing outputs from driver() is more efficient but often results in a separate file for each mesh chunk.

To block the driver routines during a call to a routine named my_finalize_routine(k), such as at the end of the computation when the drivers have no other work to do, use FEM_Update_mesh(my_finalize_routine, k, FEM_MESH_FINALIZE).

To reassemble, modify, and repartition the mesh, use FEM_Update_mesh(my_update_routine, k, FEM_MESH_UPDATE). It may be easier to perform major mesh modifications from my_update_routine(k) than the drivers, since the entire serial mesh is available to my_update_routine(k).

FEM_Update_mesh reassembles the serial mesh with an attempt to preserve the element and node global numbering. If the new mesh has the same number and type of elements and nodes, the global numbers (and hence serial mesh) will be unchanged. If new elements or nodes are added at each chunk, they will be assigned new unique global numbers. If elements or nodes are removed, their global numbers are not re-used- you can detect the resulting holes in the serial mesh since the user data associated with the deleted elements will be all zero. Generally, however, it is less error-prone to perform mesh modifications only in driver() or only in an update routine, rather than some in both.

8.8 IDXL Communication

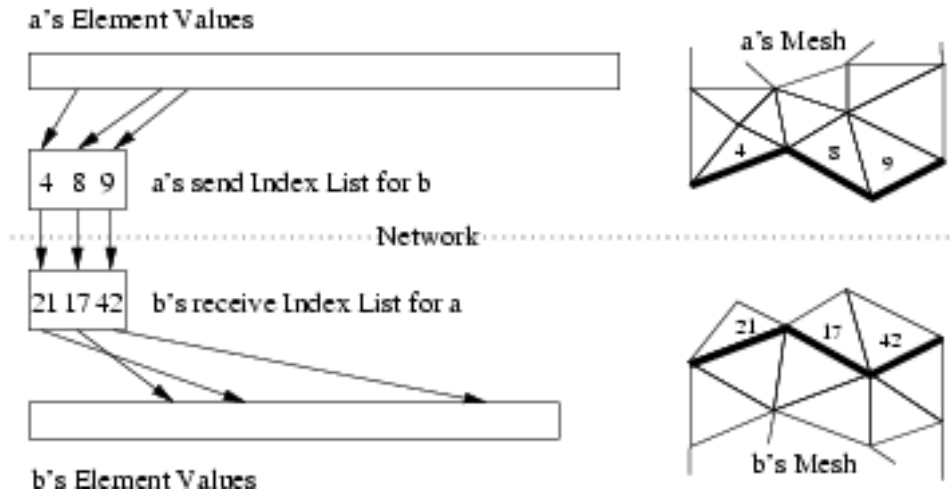
The FEM framework's communication layer is called IDXL. This small library handles sending and receiving data to and from a sparse subset of 1D indices into a user array. The sparse index subset is called an "Index List", hence the name of the library.

8.8.1 Index Lists

An Index List is the fundamental data structure of the IDXL library—for example, the list of shared nodes is an Index List. IDXL includes routines for building, combining, and sending and receiving Index Lists.

An Index List, as you might expect, is a list of indices that need to be sent and received. An Index List includes both the indices that need to be sent, as well as the indices to be received, from each chunk.

Consider two chunks a and b where b needs some information a has, such as if b has ghosts of real elements on a . a 's Index List thus has a send portion with the a -local indices for the elements a sends; and b 's Index List contains a receive portion with the b -local indices for the elements b receives. Thus across processors, the corresponding send and receive portions of a and b 's Index Lists match, as shown in Figure 15.



15: Illustrating how Index Lists match up a 's source elements with b 's ghost elements.

Index List Calls

You refer to an Index List via an opaque handle—in C, the integer typedef `IDXL_t`; in Fortran, a bare `INTEGER`.

```
IDXL_t FEM_Comm_shared(int mesh,int entity);
```

```
INTEGER function FEM_Comm_shared(mesh,entity)
INTEGER, INTENT(IN) :: mesh,entity
```

Return a read-only copy of the Index List of shared nodes. The send and receive portions of this list are identical, because each shared node is both sent and received. Shared nodes are most often used with the send/sum communication pattern.

Must be called from driver. `mesh` must be a reading mesh. `entity` must be `FEM_NODE`. You may not call `IDXL_Destroy` on the returned list.

```
IDXL_t FEM_Comm_ghost(int mesh,int entity);
```

```
INTEGER function FEM_Comm_ghost(mesh,entity)
INTEGER, INTENT(IN) :: mesh,entity
```

Return a read-only copy of the Index List of ghost entities. The send portion of this list contains real, interior entities, which are sent away; the receive portion of the list contains the ghost entities, which are received. Ghosts are most often used with the send/rcv communication pattern.

Elements to be sent out are listed starting at zero (one in Fortran); but ghost elements to be received are also listed starting at zero (one in Fortran). If real and ghost elements are kept in separate arrays, this is usable as-is; but if ghosts and real elements are kept together, you will need to shift the ghost indices using `IDXL_Combine` or `IDXL_Shift`.

This routine must be called from driver. mesh must be a reading mesh. entity must not include FEM_GHOST-ghosts are already included. You may not call `IDXL_Destroy` on the returned list.

```
IDXL_t IDXL_Create(void);
```

```
INTEGER function IDXL_Create()
```

Create a new, empty Index List. This list can then be filled up using `IDXL_Copy` or `IDXL_Combine`.

Must be called from driver. You must eventually call `IDXL_Destroy` on the returned list.

```
void IDXL_Combine(IDXL_t dest, IDXL_t src, int startSend, int startRecv);
```

```
SUBROUTINE IDXL_Combine(dest, src, startSend, startRecv)
INTEGER, INTENT(IN) :: dest, src, startSend, startRecv
```

Add the shifted contents of the src Index List to dest. The send portion of src is shifted so the first index sent will be startSend; for a ghost index list this is the index of the first sent real entity. The receive portion of src is similarly shifted so the first index received will be startRecv; for a ghost index list this is the index of the first received ghost entity.

This routine does not check for duplicates—if an index originally appears in dest and the also in the shifted src, it will be listed twice.

Advanced Index List Calls

```
void IDXL_Destroy(IDXL_t l);
```

```
SUBROUTINE IDXL_Destroy(l)
INTEGER, INTENT(IN) :: l
```

Destroy this Index List, and free the list storage allocated by the framework. Only call this routine with lists you created using `IDXL_Create`; not lists obtained directly from the FEM framework.

```
void IDXL_Print(IDXL_t l);
```

```
SUBROUTINE IDXL_Print(l)
INTEGER, INTENT(IN) :: l
```

Print out the contents of this Index List. This routine shows both the send and receive indices on the list, for each chunk we communicate with.

```
void IDXL_Copy(IDXL_t dest, IDXL_t src);
```

```
SUBROUTINE IDXL_Copy(dest, src)
INTEGER, INTENT(IN) :: dest, src
```

Copy the contents of the source Index List into the destination Index List, which should be empty.

```
void IDXL_Shift(IDXL_t l, int startSend, int startRecv);
```

```
SUBROUTINE IDXL_Shift(l,startSend,startRecv)
INTEGER, INTENT(IN) :: l,startSend,startRecv
```

Like IDXL_Combine, but only shifts the indices within a single list.

```
void IDXL_Add_entity(int newIdx,int nBetween,int *between);
```

```
SUBROUTINE IDXL_Add_node(newIdx,nBetween,between)
INTEGER, INTENT(IN) :: newIdx,nBetween
INTEGER, INTENT(IN) :: between(nBetween)
```

This call adds a new entity, with local index *newIdx*, to this Index List. The new entity is sent or received by each chunk that sends or receives all the entities listed in the *between* array. For example, when adding a new node along an edge, *nBetween* is 2 and *between* lists the endpoints of the edge; this way if the edge is shared with some chunk, the new node will be shared with that chunk.

This routine only affects the current chunk- no other chunks are affected. To ensure the communication lists match, IDXL_Add_entity must be called on all the chunks that send or receive the entity, to create the local copies of the entity.

IDXL_Add_entity adds the new entity to the end of the communication list, and so must be called in the same order on all the chunks that share the new entity. For example, if two new nodes *x* and *y* are added between chunks *a* and *b*, if chunk *a* calls IDXL_Add_entity with its local number for *x* before it calls IDXL_Add_entity with its local number for *y*, chunk *b* must also add its copy of node *x* before adding *y*.

8.8.2 Data Layout

IDXL is designed to send and receive data directly out of your arrays, with no intermediate copying. This means IDXL needs a completely general method for specifying how you store your data in your arrays. Since you probably don't change your storage layout at runtime, you can create a "data layout" once at the beginning of your program, then use it repeatedly for communication.

IDXL Layouts are normally used to describe arrays of data associated with nodes or elements. The layout abstraction allows you to use IDXL routines to communicate any sort of data, stored in a variety of formats.

Like Index Lists, Layouts are referred to via an opaque handle—in a C program via the integer typedef IDXL_Layout_t, and in Fortran via a bare integer.

Layout Routines

In most programs, the data to be communicated is a dense array of data of one type. In this case, there is only one layout routine you need to know:

```
IDXL_Layout_t IDXL_Layout_create(int type,int width);
```

```
INTEGER function IDXL_Layout_create(type,width)
INTEGER, INTENT(IN) :: type,width
```

The simplest data layout to describe—a dense array of this IDXL datatype, indexed by entity number, with width pieces of data per entity. Note that the number of entities is not stored with the layout-the number of entities to be communicated depends on the communication routine.

The IDXL datatypes are:

IDXL Datatype	C Datatypes	Fortran Datatypes
IDXL_BYTE	unsigned char	INTEGER*1
	char	LOGICAL*1
IDXL_INT	int	INTEGER
IDXL_REAL	float	SINGLE PRECISION
		REAL*4
IDXL_DOUBLE	double	DOUBLE PRECISION
		REAL*8

For example, if you keep a dense array with 3 doubles of force per node, you'd call this routine as:

```
// C++ version:
double *force=new double[3*n];
IDXL_Layout_t force_layout=IDXL_Layout_create(IDXL_DOUBLE,3);
```

```
! F90 Version
double precision, allocatable :: force(:, :)
integer :: force_layout
ALLOCATE(force(3,n)) ! (could equivalently use force(3*n) )
force_layout=IDXL_Layout_create(IDXL_DOUBLE,3)
```

This routine was once called FEM_Create_simple_field.

Advanced Layout Routines

These advanced routines are only needed if you want to exchange data stored in an array of user-defined types. Most programs only need IDXL_Layout_create.

```
IDXL_Layout_t IDXL_Layout_offset(int type, int width, int offsetBytes,
int distanceBytes, int skewBytes);
```

```
INTEGER function IDXL_Layout_offset(type,width,offsetBytes,distanceBytes,skewBytes)
INTEGER, INTENT(IN) :: type,width,offsetBytes,distanceBytes,skewBytes
```

The most general data layout—an array indexed by entity, containing width pieces of data per entity. This routine expands on IDXL_Layout_create by adding support for user-defined types or other unusual data layouts. You describe your layout by giving various in-memory byte offsets that describe the data is stored. Again, the number of entities is not stored with the layout—the number of entities to be communicated depends on the communication routine.

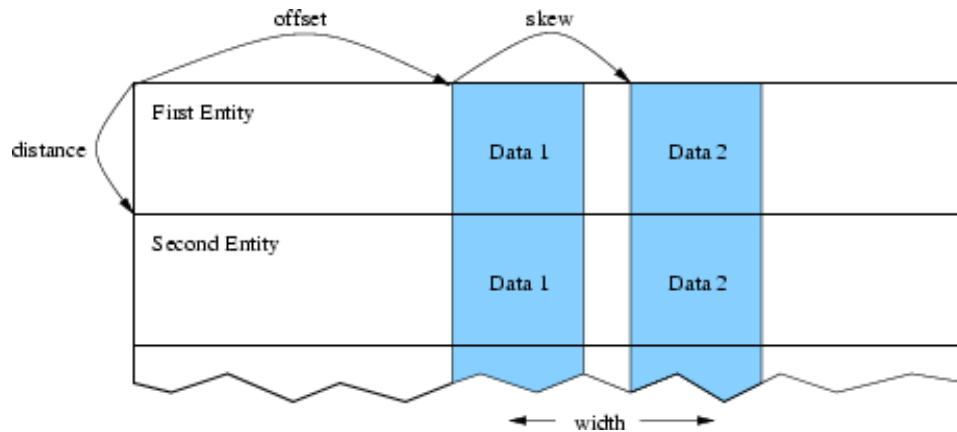
- offsetBytes The number of bytes from the start of the array to the start of the data.
- distanceBytes The number of bytes taken by one entity.
- skewBytes The number of bytes between each piece of data. Since this can almost always be determined from the size of the base data type, this parameter can be left as zero.

For example, if your node data is all stored in a struct (in fortran, a named TYPE), offsetBytes gives the distance between the start of the struct and the force; and distanceBytes gives the size in bytes of the struct.

In C, the offsetof and sizeof keywords are useful for finding these values. In Fortran, we provide a special routine called foffsetof that returns the distance, in bytes, between its two arguments.

```
// C++ version:
typedef struct {
    double d[3], v[3], force[3], a[3];
```

(continues on next page)



16: Describing a complex data layout.

(continued from previous page)

```

double m;
} node;
node *nodes=new node[n];
IDXLayout_t force_layout=IDXLayout_offset(IDXDOUBLE,3,
    offsetof(node,force),sizeof(node),0);

```

```

! F90 Version
TYPE node
  DOUBLE PRECISION :: d(3), v(3), force(3), a(3)
  DOUBLE PRECISION :: m
END TYPE
integer :: force_layout
ALLOCATE(nodes(n))
force_layout=IDXLayout_create(IDXDOUBLE,3,
&    foffsetof(nodes(1),nodes(1)%force),
&    foffsetof(nodes(1),nodes(2)),0)

```

```

void IDXLayout_destroy(IDXLayout_t layout);

```

```

SUBROUTINE IDXLayout_destroy(layout)
  INTEGER, INTENT(IN) :: layout

```

Destroy this Layout. You only need call this routine if you repeatedly create layouts.

```

int IDXLayout_get_layout_type(IDXLayout_t layout);

```

```

INTEGER function IDXLayout_get_layout_type(layout)

```

Return the IDXLayout datatype for this layout.

```

int IDXLayout_get_layout_width(IDXLayout_t layout);

```

```

INTEGER function IDXLayout_get_layout_width(layout)

```

Return the layout width—the number of data items that are communicated per entity.

```
int IDXL_Get_layout_distance(IDXL_Layout_t layout);
```

```
INTEGER function IDXL_Get_layout_distance(layout)
```

Return the layout distance—the number of bytes between successive entity’s data items.

Layout Compatibility Routines

Before IDXL was made a separate library, FEM included these routines, which are still preserved for backward compatibility.

```
IDXL_Layout_t FEM_Create_simple_field(int type,int width);
```

```
INTEGER function FEM_Create_simple_field(type,width)
INTEGER, INTENT(IN) :: type,width
```

This routine is completely interchangeable to IDXL_Layout_create.

```
int FEM_Create_field(int type,int width,int offset,int distance);
```

```
INTEGER function FEM_Create_field(type, width, offset, distance)
INTEGER, INTENT(IN) :: type, width, offset, distance
```

This routine is like a call to IDXL_Layout_offset with the rarely used skewBytes set to zero.

8.8.3 IDXL Communication

This section brings together all the pieces of IDXL: Index Lists are used to determine what to send and what to receive and Layouts are used to determine where to get and put the communicated data.

Communication Routines

```
void IDXL_Comm_sendsum(IDXL_Comm_t comm,IDXL_t indices,IDXL_Layout_t
layout,void *data);
```

```
SUBROUTINE IDXL_Comm_sendsum(comm,indices,layout,data)
INTEGER, INTENT(IN) :: comm,indices,layout
varies, INTENT(INOUT) :: data
```

Sum these indices of shared entities across all chunks that share them. The user data array is interpreted according to the given layout.

If comm is zero, this routine is blocking and finishes the communication immediately. If comm is not zero, this routine is non-blocking and equivalent to a call to IDXL_Comm_send followed by a call to IDXL_Comm_sum.

This routine is typically used to sum up partial values on shared nodes. It is a more general version of the old FEM routine FEM_Update_field. For example, to sum up the shared-node values in a 3d force vector indexed by node, you would use:

```
// C++ version:
double *force=new double[3*nNodes];
IDXLayout_t force_layout=IDXLayout_create(IDXDOUBLE,3);
IDX_t shared_indices=FEMComm_shared(mesh,FEM_NODE);

//... in the time loop ...
    IDXComm_sendsum(0,shared_indices,force_layout,force);
```

```
! F90 Version
double precision, allocatable :: force(:, :)
integer :: force_layout, shared_indices
ALLOCATE(force(3,nNodes)) ! (could equivalently use force(3*nNodes) )
force_layout=IDXLayout_create(IDXDOUBLE,3)
shared_indices=FEMComm_shared(mesh,FEM_NODE)

!... in the time loop ...
    CALL IDXComm_sendsum(0,shared_indices,force_layout,force)
```

```
void IDXComm_sendrecv(IDXComm_t comm,IDX_t indices,IDXLayout_t
layout,void *data);
```

```
SUBROUTINE IDXComm_sendrecv(comm,indices,layout,data)
INTEGER, INTENT(IN) :: comm,indices,layout
varies, INTENT(INOUT) :: data
```

Send these (typically real) send indices and copy in these (typically ghost) receive indices. The user data array is interpreted according to the given layout.

If comm is zero, this routine is blocking and finishes the communication immediately. If comm is not zero, this routine is non-blocking and equivalent to a call to `IDXLComm_send` followed by a call to `IDXLComm_sum`.

This routine is typically used to obtain the values of ghost entities. It is a more general version of the old FEM routine `FEM_Update_ghost_field`. For example, to obtain 7 solution values per ghost element, storing gElem ghosts in the array just after the nElem regular elements, we could:

```
// C++ version:
double *elem=new double[7*(nElem+gElem)];
IDXLayout_t elem_layout=IDXLayout_create(IDXDOUBLE,7);
IDX_t ghost_original=FEMComm_ghost(mesh,FEM_ELEM+1);
IDX_t ghost_shifted=IDXCreate(); // ghosts start at nElem
IDXCombine(ghost_shifted,ghost_original,0,nElem);

//... in the time loop ...
    IDXComm_sendrecv(0,ghost_shifted,elem_layout,elem);
```

```
! F90 Version
double precision, allocatable :: elem(:, :)
integer :: elem_layout, ghost_original,ghost_shifted
ALLOCATE(elem(7,nElem+gElem))
elem_layout=IDXLayout_create(IDXDOUBLE,7)
ghost_original=FEMComm_ghost(mesh,FEM_ELEM+1)
ghost_shifted=IDXCreate() ! ghosts start at nElem+1
CALL IDXCombine(ghost_shifted,ghost_original,1,nElem+1)

!... in the time loop ...
    CALL IDXComm_sendrecv(0,ghost_shifted,elem_layout,elem)
```

Advanced Communication Routines

```
IDXL_Comm_t IDXL_Comm_begin(int tag, int context);
```

```
INTEGER function IDXL_Comm_begin(tag, context)
INTEGER, INTENT(IN) :: tag, context
```

Start a non-blocking communication operation with this (user-defined) tag and communication context (0, or an AMPI communicator).

Every call to this routine must eventually be matched by a call to IDXL_Comm_wait. Warning: for now, tag and context are ignored, and there can be only one outstanding communication operation.

```
void IDXL_Comm_send(IDXL_Comm_t comm, IDXL_t indices, IDXL_Layout_t
layout, const void *data);
```

```
SUBROUTINE IDXL_Comm_send(comm, indices, layout, data)
INTEGER, INTENT(IN) :: comm, indices, layout
varies, INTENT(IN) :: data
```

When comm is flushed, send these send indices, with this layout, from this data array.

This routine is always non-blocking; as the data array passed in will not be copied out until the call to IDXL_Comm_flush.

```
void IDXL_Comm_recv(IDXL_Comm_t comm, IDXL_t indices, IDXL_Layout_t
layout, void *data);
```

```
SUBROUTINE IDXL_Comm_recv(comm, indices, layout, data)
INTEGER, INTENT(IN) :: comm, indices, layout
varies, INTENT(OUT) :: data
```

When comm is finished, copy in these receive indices, with this layout, into this data array.

This routine is always non-blocking; as the data array passed in will not be copied into until the call to IDXL_Comm_wait.

```
void IDXL_Comm_sum(IDXL_Comm_t comm, IDXL_t indices, IDXL_Layout_t
layout, void *data);
```

```
SUBROUTINE IDXL_Comm_sum(comm, indices, layout, data)
INTEGER, INTENT(IN) :: comm, indices, layout
varies, INTENT(INOUT) :: data
```

When comm is finished, add in the values for these receive indices, with this layout, into this data array.

This routine is always non-blocking; as the data array passed in will not be added to until the call to IDXL_Comm_wait.

```
void IDXL_Comm_flush(IDXL_Comm_t comm);
```

```
SUBROUTINE IDXL_Comm_flush(comm)
INTEGER, INTENT(IN) :: comm
```

Send all outgoing data listed on this comm. This routine exists because there may be many calls to IDXL_Comm_send, and sending one large message is more efficient than sending many small messages.

This routine is typically non-blocking, and may only be issued at most once per IDXL_Comm_begin.

```
void IDXL_Comm_wait(IDXL_Comm_t comm);
```

```
SUBROUTINE IDXL_Comm_wait(comm)
  INTEGER, INTENT(IN) :: comm
```

Finish this communication operation. This call must be issued exactly once per IDXL_Comm_begin. This call includes IDXL_Comm_flush if it has not yet been called.

This routine always blocks until all incoming data is received, and is the last call that can be made on this comm.

8.9 Old Communication Routines

(This section is for backward compatibility only. The IDXL routines are the new, more flexible way to perform communication.)

The FEM framework handles the updating of the values of shared nodes- that is, it combines shared nodes' values across all processors. The basic mechanism to do this update is the "field"- numeric data items associated with each node. We make no assumptions about the meaning of the node data, allow various data types, and allow a mix of communicated and non-communicated data associated with each node. The framework uses IDXL layouts to find the data items associated with each node in memory.

Each field represents a (set of) data records stored in a contiguous array, often indexed by node number. You create a field once, with the IDXL layout routines or FEM_Create_field, then pass the resulting field ID to FEM_Update_field (which does the shared node communication), FEM_Reduce_field (which applies a reduction over node values), or one of the other routines described below.

```
void FEM_Update_field(int Fid, void *nodes);
```

```
SUBROUTINE FEM_Update_field(Fid,nodes)
  INTEGER, INTENT(IN) :: Fid
  varies, INTENT(INOUT) :: nodes
```

Combine a field of all shared nodes with the other chunks. Sums the value of the given field across all chunks that share each node. For the example above, once each chunk has computed the net force on each local node, this routine will sum the net force across all shared nodes.

FEM_Update_field can only be called from driver, and to be useful, must be called from every chunk's driver routine.

After this routine returns, the given field of each shared node will be the same across all processors that share the node.

This routine is equivalent to an IDXL_Comm_Sendsum operation.

```
void FEM_Read_field(int Fid, void *nodes, char *fName);
```

```
SUBROUTINE FEM_Read_field(Fid,nodes,fName)
  INTEGER, INTENT(IN) :: Fid
  varies, INTENT(OUT) :: nodes
  CHARACTER*, INTENT(IN) :: fName
```

Read a field out of the given serial input file. The serial input file is line-oriented ASCII- each line begins with the global node number (which must match the line order in the file), followed by the data to be read into the node field. The remainder of each line is unread. If called from Fortran, the first line must be numbered 1; if called from C, the first line must be numbered zero. All fields are separated by white space (any number of tabs or spaces).

For example, if we have called Create_field to describe 3 doubles, the input file could begin with


```

1      0.2      0.7      -0.3      First node
2      0.4      1.12     -17.26     another node
...

```

FEM_Read_field must be called from driver at any time, independent of other chunks.

This routine has no IDXL equivalent.

```
void FEM_Reduce_field(int Fid, const void *nodes, void *out, int op);
```

```

SUBROUTINE FEM_Reduce_field(Fid,nodes,outVal,op)
INTEGER, INTENT(IN) :: Fid,op
varies, INTENT(IN) :: nodes
varies, INTENT(OUT) :: outVal

```

Combine one record per node of this field, according to op, across all chunks. Shared nodes are not double-counted- only one copy will contribute to the reduction. After Reduce_field returns, all chunks will have identical values in outVal, which must be vec_len copies of base_type.

May only be called from driver, and to complete, must be called from every chunk's driver routine.

op must be one of:

- FEM_SUM- each element of outVal will be the sum of the corresponding fields of all nodes
- FEM_MIN- each element of outVal will be the smallest value among the corresponding field of all nodes
- FEM_MAX- each element of outVal will be the largest value among the corresponding field of all nodes

This routine has no IDXL equivalent.

```
void FEM_Reduce(int Fid, const void *inVal, void *outVal, int op);
```

```

SUBROUTINE FEM_Reduce(Fid,inVal,outVal,op)
INTEGER, INTENT(IN) :: Fid,op
varies, INTENT(IN) :: inVal
varies, INTENT(OUT) :: outVal

```

Combine one record of this field from each chunk, according to op, across all chunks. Fid is only used for the base_type and vec_len- offset and dist are not used. After this call returns, all chunks will have identical values in outVal. Op has the same values and meaning as FEM_Reduce_field.

May only be called from driver, and to complete, must be called from every chunk's driver routine.

```

! C example
double inArr[3], outArr[3];
int fid=IDXL_Layout_create(FEM_DOUBLE,3);
FEM_Reduce(fid,inArr,outArr,FEM_SUM);

```

```

! f90 example
DOUBLE PRECISION :: inArr(3), outArr(3)
INTEGER fid
fid=IDXL_Layout_create(FEM_DOUBLE,3)
CALL FEM_Reduce(fid,inArr,outArr,FEM_SUM)

```

This routine has no IDXL equivalent.

8.9.1 Ghost Communication

It is possible to get values for a chunk's ghost nodes and elements from the neighbors. To do this, use:

```
void FEM_Update_ghost_field(int Fid, int elTypeOrMinusOne, void
*data);
```

```
SUBROUTINE FEM_Update_ghost_field(Fid,elTypeOrZero,data)
INTEGER, INTENT(IN) :: Fid,elTypeOrZero
varies, INTENT(INOUT) :: data
```

This has the same requirements and call sequence as FEM_Update_field, except it applies to ghosts. You specify which type of element to exchange using the elType parameter. Specify -1 (C version) or 0 (fortran version) to exchange node values.

8.9.2 Ghost List Exchange

It is possible to exchange sparse lists of ghost elements between FEM chunks.

```
void FEM_Exchange_ghost_lists(int elemType,int nIdx,const int
*localIdx);
```

```
SUBROUTINE FEM_Exchange_ghost_lists(elemType,nIdx,localIdx)
INTEGER, INTENT(IN) :: elemType,nIdx
INTEGER, INTENT(IN) :: localIdx[nIdx]
```

This routine sends the local element indices in localIdx to those neighboring chunks that connect to its ghost elements on the other side. That is, if the element localIdx[i] has a ghost on some chunk c, localIdx[i] will be sent to and show up in the ghost list of chunk c.

```
int FEM_Get_ghost_list_length(void);
```

Returns the number of entries in my ghost list—the number of my ghosts that other chunks passed to their call to FEM_Exchange_ghost_lists.

```
void FEM_Get_ghost_list(int *retLocalIdx);
```

```
SUBROUTINE FEM_Get_ghost_list(retLocalIdx)
INTEGER, INTENT(OUT) :: retLocalIdx[FEM_Get_ghost_list_length()]
```

These routines access the list of local elements sent by other chunks. The returned indices will all refer to ghost elements in my chunk.

8.10 ParFUM

ParFUM is the name for the latest version of FEM. ParFUM includes additional features including parallel mesh modification and adaptivity (geometrical). ParFUM also contains functions which generate additional topological adjacency information. ParFUM cannot be built separate from Charm++ since it uses various messaging mechanisms that MPI does not readily support. It is important to note that ParFUM adaptivity at the moment has some limitations. It works only for meshes which are two-dimensional. The other limitation is that the mesh on which it works on must have one layer of node-deep ghosts. Most applications require no or one layer ghosts, so it is really not a limitation, but for applications that need multiple layers of ghost information, the adaptivity operations cannot be used.

8.10.1 Adaptivity Initialization

If a FEM application wants to use parallel mesh adaptivity, the first task is to call the initialization routine from the *driver* function. This creates the node and element adjacency information that is essential for the adaptivity operations. It also initializes all the mesh adaptivity related internal objects in the framework.

```
void FEM_ADAPT_Init(int meshID)
```

Initializes the mesh defined by meshID for the mesh adaptivity operations.

8.10.2 Preparing the Mesh for Adaptivity

For every element entity in the mesh, there is a desired size entry for each element. This entry is called meshSizing. This meshSizing entry contains a metric that decides the element quality. The default metric is the average of the size of the three edges of an element. This section provides various mechanisms to set this field. Some of the adaptive operations actually use these metrics to maintain quality. Though there is another metric which is computer for each element and maintained on the fly and that is the ratio of the largest length to the smallest altitude and this value during mesh adaptivity is not allowed to go beyond a certain limit. Because the larger this value after a certain limit, the worse the element quality.

```
void FEM_ADAPT_SetElementSizeField(int meshID, int elem, double size);
```

For the mesh specified by meshID, for the element elem, we set the desired size for each element to be size.

```
void FEM_ADAPT_SetElementSizeField(int meshID, double *sizes);
```

For the mesh specified by meshID, for the element elem, we set the desired size for each element from the corresponding entry in the sizes array.

```
void FEM_ADAPT_SetReferenceMesh(int meshID);
```

For each element int this mesh defined by meshID set its size to the average edge length of the corresponding element.

```
void FEM_ADAPT_GradateMesh(int meshID, double smoothness);
```

Resize mesh elements to avoid jumps in element size. i.e. avoid discontinuities in the desired sizes for elements of a mesh by smoothing them out. Algorithm based on h-shock correction, described in Mesh Gradation Control, Borouchaki et al.

8.10.3 Modifying the Mesh

Once the elements in the mesh has been prepared by specifying their desired sizes, we are ready to use the actual adaptivity operations. Currently we provide Delaunay flip operations, edge bisect operations and edge-coarsen operations all of which are implemented in parallel, but the user has access to these wrapper functions which intelligently decide when and in which region of the mesh to use the adaptivity operations to generate a mesh with higher quality elements while achieving the desired size (which is usually average edge length per element), or it could even be the area of each element.

```
void FEM_ADAPT_Refine(int meshID, int qm, int method, double factor, double *sizes);
```

Perform refinements on the mesh specified by meshId. Tries to maintain/improve element quality by refining the mesh as specified by a quality measure qm. If method = 0, refine areas with size larger than factor down to factor If method

= 1, refine elements down to sizes specified in the sizes array. In this array each entry corresponds to the corresponding element. Negative entries in sizes array indicate no refinement.

```
void FEM_ADAPT_Coarsen(int meshID, int qm, int method, double
factor, double *sizes);
```

Perform refinements on the mesh specified by meshID. Tries to maintain/improve element quality by coarsening the mesh as specified by a quality measure qm. If method = 0, coarsen areas with size smaller than factor down to factor. If method = 1, coarsen elements up to sizes specified in the sizes array. In this array each entry corresponds to the corresponding element. Negative entries in sizes array indicate no coarsening.

```
void FEM_ADAPT_AdaptMesh(int meshID, int qm, int method, double
factor, double *sizes);
```

It has the same set of arguments as required by the previous two operations, namely refine and coarsen. This function keeps using the above two functions till we have all elements in the mesh with as close to the desired quality. Apart from using the above two operations, it also performs a mesh repair operation where it gets rid of some bad quality elements by Delaunay flip or coarsening as the geometry in the area demands.

```
int FEM_ADAPT_SimpleRefineMesh(int meshID, double targetA, double xmin,
double ymin, double xmax, double ymax);
```

A region is defined by (xmax, xmin, ymax, ymin) and the target area to be achieved for all elements in this region in the mesh specified by meshID is given by targetA. This function only performs a series of refinements on the elements in this region. If the area is larger, then no coarsening is done.

```
int FEM_ADAPT_SimpleCoarsenMesh(int meshID, double targetA, double xmin,
double ymin, double xmax, double ymax);
```

A region is defined by (xmax, xmin, ymax, ymin) and the target area to be achieved for all elements in this region in the mesh specified by meshID is given by targetA. This function only performs a series of coarsenings on the elements in this region. If the area is smaller, then no refinement is done.

8.10.4 Verify correctness of the Mesh

After adaptivity operations are performed and even before adaptivity operations, it is important to first verify that we are working on a mesh that is consistent geometrically with the types of mesh that the adaptivity algorithms are designed to work on. There is a function that can be used to test various properties of a mesh, like area, quality, geometric consistency, idxl list correctness, etc.

```
void FEM_ADAPT_TestMesh(int meshID);
```

This provides a series of tests to determine the consistency of the mesh specified by meshID.

These four simple steps define what needs to be used by a program that wishes to use the adaptivity features of ParFUM.

8.10.5 ParFUM developers

This manual is meant for ParFUM users, so developers should look at the source code and the doxygen generated documentation.

Iterative Finite Element Matrix (IFEM) Framework

Contents

- *Iterative Finite Element Matrix (IFEM) Framework*
 - *Introduction*
 - * *Terminology*
 - *Solvers*
 - * *Conjugate Gradient Solver*
 - *Solving Shared-Node Systems*
 - * *IFEM_Solve_shared*
 - * *IFEM_Solve_shared_bc*

9.1 Introduction

This manual presents the Iterative Finite Element Matrix (IFEM) library, a library for easily solving matrix problems derived from finite-element formulations. The library is designed to be matrix-free, in that the only matrix operation required is matrix-vector product, and hence the entire matrix need never be assembled.

IFEM is built on the mesh and communication capabilities of the Charm++ FEM Framework, so for details on the basic runtime, problem setup, and partitioning see the FEM Framework manual.

9.1.1 Terminology

A FEM program manipulates elements and nodes. An **element** is a portion of the problem domain, also known as a cell, and is typically some simple shape like a triangle, square, or hexagon in 2D; or tetrahedron or rectangular solid

in 3D. A **node** is a point in the domain, and is often the vertex of several elements. Together, the elements and nodes form a **mesh**, which is the central data structure in the FEM framework. See the FEM manual for details.

9.2 Solvers

A IFEM **solver** is a subroutine that controls the search for the solution.

Solvers often take extra parameters, which are listed in a type called in C `ILSI_Param`, which in Fortran is an array of `ILSI_PARAM` doubles. You initialize these solver parameters using the subroutine `ILSI_Param_new`, which takes the parameters as its only argument. The input and output parameters in an `ILSI_Param` are listed in Table 8 and Table 9.

8: `ILSI_Param` solver input parameters.

C Field Name	Fortran Field Offset	Use
maxResidual	1	If nonzero, termination criteria: magnitude of residual.
maxIterations	2	If nonzero, termination criteria: number of iterations.
solverIn[8]	3-10	Solver-specific input parameters.

9: `ILSI_Param` solver output parameters.

C Field Name	Fortran Field Offset	Use
residual	11	Magnitude of residual of final solution.
iterations	12	Number of iterations actually taken.
solverOut[8]	13-20	Solver-specific output parameters.

9.2.1 Conjugate Gradient Solver

The only solver currently written using IFEM is the conjugate gradient solver. This linear solver requires the matrix to be real, symmetric and positive definite.

Each iteration of the conjugate gradient solver requires one matrix-vector product and two global dot products. For well-conditioned problems, the solver typically converges in some small multiple of the diameter of the mesh-the number of elements along the largest side of the mesh.

You access the conjugate gradient solver via the subroutine name `ILSI_CG_Solver`.

9.3 Solving Shared-Node Systems

Many problems encountered in FEM analysis place the entries of the known and unknown vectors at the nodes-the vertices of the domain. Elements provide linear relationships between the known and unknown node values, and the entire matrix expresses the combination of all these element relations.

For example, in a structural statics problem, we know the net force at each node, f , and seek the displacements of each node, u . Elements provide the relationship, often called a stiffness matrix K , between a nodes' displacements and its net forces:

$$f = Ku$$

We normally label the known vector b (in the example, the forces), the unknown vector x (in the example, the displacements), and the matrix A :

$$b = Ax$$

IFEM provides two routines for solving problems of this type. The first routine, `IFEM_Solve_shared`, solves for the entire x vector based on the known values of the b vector. The second, `IFEM_Solve_shared_bc`, allows certain entries in the x vector to be given specific values before the problem is solved, creating values for the b vector.

9.3.1 IFEM_Solve_shared

```
void IFEM_Solve_shared(ILSI_Solver s, ILSI_Param *p, int fem_mesh, int
    fem_entity, int length, int width, IFEM_Matrix_product_c A, void *ptr,
    const double *b, double *x);
```

```
subroutine IFEM_Solve_shared(s, p, fem_mesh, fem_entity, length, width, A, ptr, b, x)
external solver subroutine :: s
double precision, intent(inout) :: p(ILSI_PARAM)
integer, intent(in) :: fem_mesh, fem_entity, length, width
external matrix-vector product subroutine :: A
TYPE(varies), pointer :: ptr
double precision, intent(in) :: b(width,length)
double precision, intent(inout) :: x(width,length)
```

This routine solves the linear system $Ax = b$ for the unknown vector x . s and p give the particular linear solver to use, and are described in more detail in Section 9.2. `fem_mesh` and `fem_entity` give the FEM framework mesh (often `FEM_Mesh_default_read()`) and entity (often `FEM_NODE`) with which the known and unknown vectors are listed.

`width` gives the number of degrees of freedom (entries in the vector) per node. For example, if there is one degree of freedom per node, `width` is one. `length` should always equal the number of FEM nodes.

A is a local matrix-vector product routine you must write. Its interface is described in Section 9.3.1. ptr is a pointer passed down to A —it is not otherwise used by the framework.

b is the known vector. x , on input, is the initial guess for the unknown vector. On output, x is the final value for the unknown vector. b and x should both have `length * width` entries. In C, DOF i of node n should be indexed as $x[n*width+i]$. In Fortran, these arrays should be allocated like `x(width,length)`.

When this routine returns, x is the final value for the unknown vector, and the output values of the solver parameters p will have been written.

```
// C++ Example
int mesh=FEM_Mesh_default_read();
int nNodes=FEM_Mesh_get_length(mesh,FEM_NODE);
int width=3; //A 3D problem
ILSI_Param solverParam;
struct myProblemData myData;

double *b=new double[nNodes*width];
double *x=new double[nNodes*width];
... prepare solution target b and guess x ...

ILSI_Param_new(&solverParam);
solverParam.maxResidual=1.0e-4;
solverParam.maxIterations=500;

IFEM_Solve_shared(IFEM_CG_Solver,&solverParam,
    mesh,FEM_NODE,nNodes,width,
    myMatrixVectorProduct,&myData,b,x);
```

```

! F90 Example
include 'ifemf.h'
INTEGER :: mesh, nNodes, width
DOUBLE PRECISION, ALLOCATABLE :: b(:, :), x(:, :)
DOUBLE PRECISION :: solverParam(ILSI_PARAM)
TYPE(myProblemData) :: myData

mesh=FEM_Mesh_default_read()
nNodes=FEM_Mesh_get_length(mesh, FEM_NODE)
width=3    ! A 3D problem

ALLOCATE(b(width, nNodes), x(width, nNodes))
... prepare solution target b and guess x ..

ILSI_Param_new(&solverParam);
solverParam(1)=1.0e-4;
solverParam(2)=500;

IFEM_Solve_shared(IFEM_CG_Solver, solverParam,
    mesh, FEM_NODE, nNodes, width,
    myMatrixVectorProduct, myData, b, x);

```

Matrix-vector product routine

IFEM requires you to write a matrix-vector product routine that will evaluate Ax for various vectors x . You may use any subroutine name, but it must take these arguments:

```

void IFEM_Matrix_product(void *ptr, int length, int width, const double
    *src, double *dest);

```

```

subroutine IFEM_Matrix_product(ptr, length, width, src, dest)
TYPE(varies), pointer :: ptr
integer, intent(in) :: length, width
double precision, intent(in) :: src(width, length)
double precision, intent(out) :: dest(width, length)

```

The framework calls this user-written routine when it requires a matrix-vector product. This routine should compute $dest = A src$, interpreting src and $dest$ as vectors. $length$ gives the number of nodes and $width$ gives the number of degrees of freedom per node, as above.

In writing this routine, you are responsible for choosing a representation for the matrix A . For many problems, there is no need to represent A explicitly—instead, you simply evaluate $dest$ by looping over local elements, taking into account the values of src . This example shows how to write the matrix-vector product routine for simple 1D linear elastic springs, while solving for displacement given net forces.

After calling this routine, the framework will handle combining the overlapping portions of these vectors across processors to arrive at a consistent global matrix-vector product.

```

// C++ Example
#include "ifemC.h"

typedef struct {
    int nElements; //Number of local elements
    int *conn; // Nodes adjacent to each element: 2*nElements entries
    double k; //Uniform spring constant

```

(continues on next page)

(continued from previous page)

```

} myProblemData;

void myMatrixVectorProduct(void *ptr, int nNodes, int dofPerNode,
                           const double *src, double *dest)
{
    myProblemData *d=(myProblemData *)ptr;
    int n,e;
    // Zero out output force vector:
    for (n=0;n<nNodes;n++) dest[n]=0;
    // Add in forces from local elements
    for (e=0;e<d->nElements;e++) {
        int n1=d->conn[2*e+0]; // Left node
        int n2=d->conn[2*e+1]; // Right node
        double f=d->k * (src[n2]-src[n1]); //Force
        dest[n1]+=f;
        dest[n2]-=f;
    }
}

```

```

! F90 Example
TYPE(myProblemData)
    INTEGER :: nElements
    INTEGER, ALLOCATABLE :: conn(2,:)
    DOUBLE PRECISION :: k
END TYPE

SUBROUTINE myMatrixVectorProduct(d,nNodes,dofPerNode,src,dest)
    include 'ifemf.h'
    TYPE(myProblemData), pointer :: d
    INTEGER :: nNodes,dofPerNode
    DOUBLE PRECISION :: src(dofPerNode,nNodes), dest(dofPerNode,nNodes)
    INTEGER :: e,n1,n2
    DOUBLE PRECISION :: f

    dest(:, :)=0.0
    do e=1,d%nElements
        n1=d%conn(1,e)
        n2=d%conn(2,e)
        f=d%k * (src(1,n2)-src(1,n1))
        dest(1,n1)=dest(1,n1)+f
        dest(1,n2)=dest(1,n2)+f
    end do
END SUBROUTINE

```

9.3.2 IFEM_Solve_shared_bc

```

void IFEM_Solve_shared_bc(ILSI_Solver s, ILSI_Param *p, int fem_mesh,
int fem_entity, int length, int width, int bcCount, const int *bcDOF,
const double *bcValue, IFEM_Matrix_product_c A, void *ptr, const
double *b, double *x);

```

```

subroutine IFEM_Solve_shared_bc(s, p, fem_mesh, fem_entity, length, width,
bcCount, bcDOF, bcValue, A, ptr, b, x)
external solver subroutine :: s

```

(continues on next page)

(continued from previous page)

```

double precision, intent(inout) :: p(ILSI_PARAM)
integer, intent(in) :: fem_mesh, fem_entity, length,width
integer, intent(in) :: bcCount
integer, intent(in) :: bcDOF(bcCount)
double precision, intent(in) :: bcValue(bcCount)
external matrix-vector product subroutine :: A
TYPE(varies), pointer :: ptr
double precision, intent(in) :: b(width,length)
double precision, intent(inout) :: x(width,length)

```

Like IFEM_Solve_shared, this routine solves the linear system $Ax = b$ for the unknown vector x . This routine, however, adds support for boundary conditions associated with x . These so-called “essential” boundary conditions restrict the values of some unknowns. For example, in structural dynamics, a fixed displacement is such an essential boundary condition.

The only form of boundary condition currently supported is to impose a fixed value on certain unknowns, listed by their degree of freedom—that is, their entry in the unknown vector. In general, the i ’th DOF of node n has DOF number $n * width + i$ in C and $(n - 1) * width + i$ in Fortran. The framework guarantees that, on output, for all $bcCount$ boundary conditions, $x(bcDOF(f)) = bcValue(f)$.

For example, if $width$ is 3 in a 3d problem, we would set node ny ’s y coordinate to 4.6 and node nz ’s z coordinate to 7.3 like this:

```

// C++ Example
int bcCount=2;
int bcDOF[bcCount];
double bcValue[bcCount];
// Fix node ny's y coordinate
bcDOF[0]=ny*width+1; // y is coordinate 1
bcValue[0]=4.6;
// Fix node nz's z coordinate
bcDOF[1]=nz*width+2; // z is coordinate 2
bcValue[1]=2.0;

```

```

! F90 Example
integer :: bcCount=2;
integer :: bcDOF(bcCount);
double precision :: bcValue(bcCount);
// Fix node ny's y coordinate
bcDOF(1)=(ny-1)*width+2; // y is coordinate 2
bcValue(1)=4.6;
// Fix node nz's z coordinate
bcDOF(2)=(nz-1)*width+3; // z is coordinate 3
bcValue(2)=2.0;

```

Mathematically, what is happening is we are splitting the partially unknown vector x into a completely unknown portion y and a known part f :

$$Ax = b$$

$$A(y + f) = b$$

$$Ay = b - Af$$

We can then define a new right hand side vector $c = b - Af$ and solve the new linear system $Ay = c$ normally. Rather than renumbering, we do this by zeroing out the known portion of x to make y . The creation of the new linear system, and the substitution back to solve the original system are all done inside this subroutine.

One important missing feature is the ability to specify general linear constraints on the unknowns, rather than imposing specific values.

Contents

- *NetFEM Framework*
 - *Introduction*
 - *Compiling and Installing*
 - *Running NetFEM Online*
 - *Running NetFEM Offline*
 - *NetFEM with other Visualization Tools*
 - *Interface Basics*
 - *Simple Interface*
 - *Advanced “Field” Interface*

10.1 Introduction

NetFEM was built to provide an easy way to visualize the current state of a finite-element simulation, or any parallel program that computes on an unstructured mesh. NetFEM is designed to require very little effort to add to a program, and connects to the running program over the network via the network protocol CCS (Converse Client/Server).

10.2 Compiling and Installing

NetFEM is part of Charm++, so it can be downloaded as part of charm. To build NetFEM, just build FEM normally, or else do a make in charm/netlrts-linux-x86_64/tmp/libs/ck-lib/netfem/.

To link with NetFEM, add `-module netfem` to your program's link line. Note that you do *not* need to use the FEM framework to use NetFEM.

The `netfem` header file for C is called `"netfem.h"`, the header for fortran is called `'netfemf.h'`. A simple example NetFEM program is in `charm/pgms/charm++/fem/simple2D/`. A more complicated example is in `charm/pgms/charm++/fem/crack2D/`.

10.3 Running NetFEM Online

Once you have a NetFEM program, you can run it and view the results online by starting the program with CCS enabled:

```
foo.bar.edu> ./charmrun ./myprogram +p2 ++server ++server-port 1234
```

`"++server-port"` controls the TCP port number to use for CCS—here, we use 1234. Currently, NetFEM only works with one chunk per processor—that is, the `-vp` option cannot be used.

To view the results online, you then start the NetFEM client, which can be downloaded for Linux or Windows from <http://charm.cs.illinois.edu/research/fem/netfem/>.

Enter the name of the machine running `charmrun` and the TCP port number into the NetFEM client—for example, you might run:

```
netfem foo.bar.edu:1234
```

The NetFEM client will then connect to the program, download the most recent mesh registered with `NetFEM_POINTAT`, and display it. At any time, you can press the `"update"` button to reload the latest mesh.

10.4 Running NetFEM Offline

Rather than using CCS as above, you can register your meshes using `NetFEM_WRITE`, which makes the server write out binary output dump files. For example, to view timestep 10, which is written to the `"NetFEM/10/"` directory, you'd run the client program as:

```
netfem NetFEM/10
```

In offline mode, the `"update"` button fetches the next extant timestep directory.

10.5 NetFEM with other Visualization Tools

You can use a provided converter program to convert the offline NetFEM files into an XML format compatible with the powerful offline visualization tool ParaView (<http://paraview.org>). The converter is located in `.../charm/src/libs/cklibs/netfem/ParaviewConverter/`. Build the converter by simply issuing a `"make"` command in that directory (assuming NetFEM already has been built).

Run the converter from the parent directory of the `"NetFEM"` directory to be converted. The converter will generate a directory called `"ParaViewData"`, which contains subdirectories for each timestep, along with a `"timestep"` directory for index files for each timestep. All files in the `ParaViewData` directory can be opened by ParaView. To open all chunks for a given timestep, open the desired timestep file in `"ParaViewData/timesteps"`. Also, individual partition files can also be opened from `"ParaViewData / <timestep> / <partition_num>"`.

10.6 Interface Basics

You publish your data via NetFEM by making a series of calls to describe the current state of your data. There are only 6 possible calls you can make.

NetFEM_Begin is the first routine you call. NetFEM_End is the last routine to call. These two calls bracket all the other NetFEM calls.

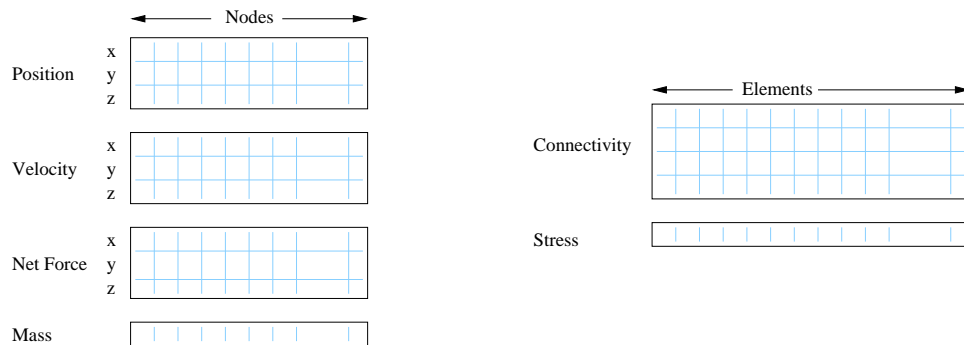
NetFEM_Nodes describes the properties of the nodes, or vertices of the domain. NetFEM_Elements describes the properties of your elements (triangles, tetrahedra, etc.). After making one of these calls, you list the different data arrays associated with your nodes or elements by making calls to NetFEM_Scalar or NetFEM_Vector.

For example, a typical finite element simulation might have a scalar mass and vector position, velocity, and net force associated with each node; and have a scalar stress value associated with each element. The sequence of NetFEM calls this application would make would be:

```
NetFEM_Begin
  NetFEM_Nodes -- lists position of each node
  NetFEM_Vector -- lists velocity of each node
  NetFEM_Vector -- lists net force on each node
  NetFEM_Scalar -- lists mass of each node

  NetFEM_Elements -- lists the nodes of each element
  NetFEM_Scalar -- lists the stress of each element

NetFEM_End
```



17: These arrays, typical of a finite element analysis program, might be passed into NetFEM.

10.7 Simple Interface

The details of how to make each call are:

```
NetFEM NetFEM_Begin(int source, int step, int dim, int flavor);
```

```
integer function NetFEM_Begin(source, step, dim, flavor)
  integer, intent(in) :: source, step, dim, flavor
```

Begins describing a single piece of a mesh. Returns a handle that is used for each subsequent call until NetFEM_End. This call, like all NetFEM calls, is collective—every processor should make the same calls in the same order.

source identifies the piece of the mesh—use FEM_My_partition or CkMyPe.

step identifies which version of the mesh this is—for example, you might use the timestep number. This is only used to identify the mesh in the client.

dim is the number of spatial dimensions. For example, in a 2D computation, you’d pass dim==2; in a 3D computation, dim==3. The client currently only supports 2D or 3D computations.

flavor specifies what to do with the data. This can take the value NetFEM_POINTAT, which is used in online visualization, and specifies that NetFEM should only keep a pointer to your data rather than copy it out of your arrays. Or it can take the value NetFEM_WRITE, which writes out the data to files named “NetFEM/step/source.dat” for offline visualization.

```
void NetFEM_End(NetFEM n);
```

```
subroutine NetFEM_End(n)
  integer, intent(in) :: n
```

Finishes describing a single piece of a mesh, which then makes the mesh available for display.

```
void NetFEM_Nodes(NetFEM n, int nNodes, const double *loc, const char *name);
```

```
subroutine NetFEM_Nodes(n, nNodes, loc, name)
  integer, intent(in) :: n, nNodes
  double precision, intent(in) :: loc(dim, nNodes)
  character*(*), intent(in) :: name
```

Describes the nodes in this piece of the mesh.

n is the NetFEM handle obtained from NetFEM_Begin.

nNodes is the number of nodes listed here.

loc is the location of each node. This must be double-precision array, laid out with the same number of dimensions as passed to NetFEM_Begin. For example, in C the location of a 2D node *n* is stored in loc[2*n+0] (x coordinate) and loc[2*n+1] (y coordinate). In Fortran, location of a node *n* is stored in loc(:,n).

name is a human-readable name for the node locations to display in the client. We recommend also including the location units here, for example “Position (m)”.

```
void NetFEM_Elements(NetFEM n, int nElements, int nodePerEl, const int *conn, const char_
↪ *name);
```

```
subroutine NetFEM_Elements(n, nElements, nodePerEl, conn, name)
  integer, intent(in) :: n, nElements, nodePerEl
  integer, intent(in) :: conn(nodePerEl, nElements)
  character*(*), intent(in) :: name
```

Describes the elements in this piece of the mesh. Unlike NetFEM_Nodes, this call can be repeated if there are different types of elements (For example, some meshes contain a mix of triangles and quadrilaterals).

n is the NetFEM handle obtained from NetFEM_Begin.

nElements is the number of elements listed here.

nodePerEl is the number of nodes for each element. For example, a triangle has 3 nodes per element; while tetrahedra have 4.

conn gives the index of each element’s nodes. Note that when called from C, the first node is listed in conn as 0 (0-based node indexing), and element *e*’s first node is stored in conn[e*nodePerEl+0]. When called from Fortran, the first node is listed as 1 (1-based node indexing), and element *e*’s first node is stored in conn(1,e) or conn((e-1)*nodePerEl+1).

name is a human-readable name for the elements to display in the client. For example, this might be “Linear-Strain Triangles”.

```
void NetFEM_Vector (NetFEM n, const double *data, const char *name);
```

```
subroutine NetFEM_Vector(n, data, name)
  integer, intent(in) :: n
  double precision, intent(in) :: data(dim, lastEntity)
  character*(*), intent(in) :: name
```

Describes a spatial vector associated with each node or element in the mesh. Attaches the vector to the most recently listed node or element. You can repeat this call several times to describe different vectors.

n is the NetFEM handle obtained from NetFEM_Begin.

data is the double-precision array of vector values. The dimensions of the array have to match up with the node or element the data is associated with-in C, a 2D element e ’s vector starts at `data[2*e]`; in Fortran, element e ’s vector is `data(:,e)`.

name is a human-readable name for this vector data. For example, this might be “Velocity (m/s)”.

```
void NetFEM_Scalar (NetFEM n, const double *data, int dataPer, const char *name);
```

```
subroutine NetFEM_Scalar(n, data, dataPer, name)
  integer, intent(in) :: n, dataPer
  double precision, intent(in) :: data(dataPer, lastEntity)
  character*(*), intent(in) :: name
```

Describes some scalar data associated with each node or element in the mesh. Like NetFEM_Vector, this data is attached to the most recently listed node or element and this call can be repeated. For a node or element, you can make the calls to NetFEM_Vector and NetFEM_Scalar in any order.

n is the NetFEM handle obtained from NetFEM_Begin.

data is the double-precision array of values. In C, an element e ’s scalar values start at `data[dataPer*e]`; in Fortran, element e ’s values are in `data(:,e)`.

dataPer is the number of values associated with each node or element. For true scalar data, this is 1; but can be any value. Even if dataPer happens to equal the number of dimensions, the client knows that this data does not represent a spatial vector.

name is a human-readable name for this scalar data. For example, this might be “Mass (Kg)” or “Stresses (pure)”.

10.8 Advanced “Field” Interface

This more advanced interface can be used if you store your node or element data in arrays of C structs or Fortran TYPES. To use this interface, you’ll have to provide the name of your struct and field. Each “field” routine is just an extended version of a regular NetFEM call described above, and can be used in place of the regular NetFEM call. In each case, you pass a description of your field in addition to the usual NetFEM parameters.

In C, use the macro “NetFEM_Field(theStruct,theField)” to describe the FIELD. For example, to describe the field “loc” of your structure named “node_t”,

```
node_t *myNodes=...;
..., NetFEM_Field(node_t, loc), ...
```

In Fortran, you must pass as FIELD the byte offset from the start of the structure to the start of the field, then the size of the structure. The FEM “foffsetof” routine, which returns the number of bytes between its arguments, can be used for this. For example, to describe the field “loc” of your named type “NODE”,

```
TYPE (NODE), ALLOCATABLE :: n(:)
..., foffsetof(n(1),n(1)%loc),foffsetof(n(1),n(2)), ...
```

```
void NetFEM_Nodes_field(NetFEM n,int nNodes,FIELD,const void *loc,const char *name);
```

```
subroutine NetFEM_Nodes_field(n,nNodes,FIELD,loc,name)
```

A FIELD version of NetFEM_Nodes.

```
void NetFEM_Elements_field(NetFEM n,int nElements,int nodePerEl,FIELD,int idxBase,
↳const int *conn,const char *name);
```

```
subroutine NetFEM_Elements_field(n,nElements,nodePerEl,FIELD,idxBase,conn,name)
```

A FIELD version of NetFEM_Elements. This version also allows you to control the starting node index of the connectivity array—in C, this is normally 0; in Fortran, this is normally 1.

```
void NetFEM_Vector_field(NetFEM n,const double *data,FIELD,const char *name);
```

```
subroutine NetFEM_Vector_field(n,data,FIELD,name)
```

A FIELD version of NetFEM_Vector.

```
void NetFEM_Scalar_field(NetFEM n,const double *data,int dataPer,FIELD,const char_
↳*name);
```

```
subroutine NetFEM_Scalar(n,data,dataPer,FIELD,name)
```

A FIELD version of NetFEM_Scalar.

Contents

- *Multiblock Framework*
 - *Motivation*
 - *Introduction/Terminology*
 - *Input Files*
 - *Structure of a Multiblock Framework Program*
 - *Compilation and Execution*
 - *Preparing Input Files*
 - *Multiblock Framework API Reference*
 - * *Initialization*
 - * *Utility*
 - * *Internal Boundary Conditions and Block Fields*
 - * *External Boundary Conditions*
 - * *Migration*

11.1 Motivation

A large class of problems can be solved by first decomposing the problem domain into a set of structured grids. For simplicity, each structured grid is often made rectangular, when it is called a *block*. These blocks may face one another or various parts of the outside world, and taken together comprise a *multiblock computation*.

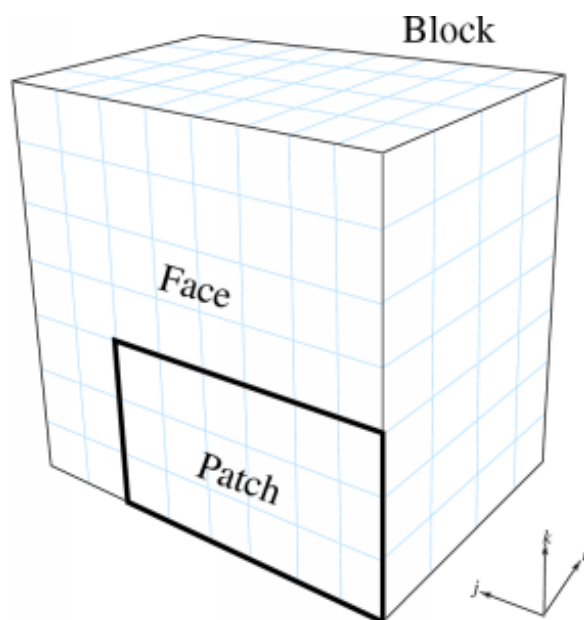
There are two main types of multiblock computations – implicit and explicit. In an implicit computation, a global

matrix, which represents the entire problem domain, is formed and solved. Implicit computations require a fast sparse matrix solver, and are typically used for steady-state problems. In an explicit computation, the solution proceeds locally, computing new values based on the values of nearby points. Explicit computations often have stability criteria, and are typically used for time-dependent problems.

The Charm++ multiblock framework allows you to write a parallel explicit multiblock program, in C or Fortran 90, by concentrating on what happens to a single block of the domain. Boundary condition housekeeping and “ghost cell” exchange are all handled transparently by the framework. Using the multiblock framework also allows you to take advantage of all the features of Charm++, including adaptive computation and communication overlap, run-time load balancing, performance monitoring and visualization, and checkpoint/restart, with no additional effort.

11.2 Introduction/Terminology

A *block* is a distorted rectangular grid that represents a portion of the problem domain. A volumetric cell in the grid is called a *voxel*. Each exterior side of a block is called a *face*. Each face may consist of several rectangular *patches*, which all abut the same block and experience the same boundary conditions.



18: Terminology used by the framework.

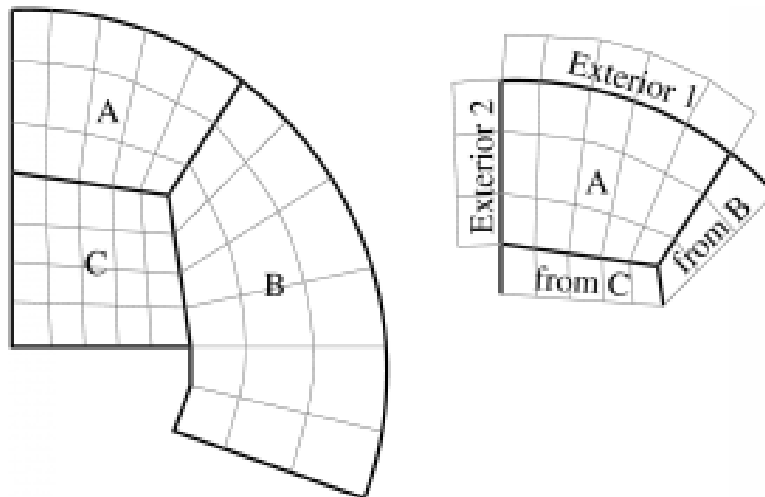
For example, Figure 18 shows a 3D 4x8x7-voxel block, with a face and 6x3 patch indicated.

The computational domain is tiled with such blocks, which are required to be conformal – the voxels must match exactly. The blocks need not be the same size or orientation, however, as illustrated in the 2D domain of Figure 19.

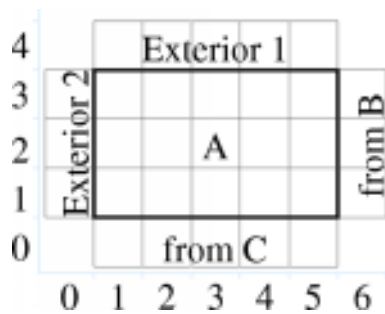
Figure 19 also shows the computation from the point of view of block A, which has two external boundary conditions (on the left and top sides) and two “internal” boundary conditions (on the right and bottom sides). During the computation, the external boundary conditions can be imposed independent of any other blocks; while the internal boundary conditions must be obtained from the other blocks.

To simplify the computation on the interior, these boundary conditions are typically written into special extra “ghost” (or dummy) cells around the outside of the real interior cells. The array indexing for these ghost cells is illustrated in Figure 20.

The Multiblock framework manages all the boundary conditions – both internal and external. Internal boundary conditions are sent across processors, and require you to register the data “fields” you wish exchanged. External



19: A 2D domain decomposed into three blocks: A (5x3), B (3x6), and C (5x4). Also shows the computation as seen from block A.



20: The ghost cells around a 5x3-voxel 2D block

boundary conditions are not communicated, but require you to register a function to apply that boundary condition to your data. Either type of boundary condition can have arbitrary thickness.

Finally, the Multiblock framework manages nothing *but* boundary conditions. The rest of the computation, such as deciding on and implementing timestepping, stencils, numerics, and interpolation schemes are all left up to the user.

11.3 Input Files

The Multiblock framework reads, in parallel, a partitioned set of blocks from block input files. Each block consists of a file with extension “.mblk” for the interior data (grid coordinates and initial conditions) and “.bblk” for the boundary condition data (patches where boundaries should be applied).

These block files are generated with a separate, offline tool called “makemblock”, which is documented elsewhere.

11.4 Structure of a Multiblock Framework Program

A Multiblock framework program consists of several subroutines: `init`, `driver`, `finalize`, and external boundary condition subroutines.

`init` and `finalize` are called by the Multiblock framework only on the first processor – these routines typically do specialized I/O, startup and shutdown tasks.

A separate driver subroutine runs for each block, doing the main work of the program. Because there may be several blocks per processor, several driver routines may execute as threads simultaneously.

The boundary condition subroutines are called by the framework after a request from the driver.

```
subroutine init
    read configuration data
end subroutine

subroutine bc1
    apply first type of boundary condition
end subroutine bc1

subroutine bc2
    apply second type of boundary condition
end subroutine bc2

subroutine driver
    allocate and initialize the grid
    register boundary condition subroutines bc1 and bc2
    time loop
        apply external boundary conditions
        apply internal boundary conditions
        perform serial internal computation
    end time loop
end subroutine

subroutine finalize
    write results
end subroutine
```

11.5 Compilation and Execution

A Multiblock framework program is a Charm++ program, so you must begin by downloading the latest source version of Charm++ from <https://charm.cs.illinois.edu>. Build the source with `./build MBLOCK version` or `cd` into the build directory, `<version>/tmp`, and type `make MBLOCK`. To compile a MULTIBLOCK program, pass the `-language mblock` (for C) or `-language mblockf` (for Fortran) option to `charmcc`.

In a charm installation, see `charm/<version>/pgms/charm++/mblock/` for example and test programs.

11.6 Preparing Input Files

The Multiblock framework reads its description of the problem domain from input “block” files, which are in a Multiblock-specific format. The files are named with the pattern `prefixnumber.ext`, where `prefix` is a arbitrary string prefix you choose, `number` is the number of this block (virtual processor), and `ext` is either “`mblk`”, which contains binary data with the block coordinates, or “`bblk`”, which contains ASCII data with the block’s boundary conditions.

You generate these Multiblock input files using a tool called *makemblock*, which can be found in `charm/<version>/pgms/charm++/makemblock`. *makemblock* can read a description of the problem domain generated by the structured meshing program Gridgen (from Pointwise) in `.grd` and `.inp` format; or read a binary `.msh` format. *makemblock* divides this input domain into the number of blocks you specify, then writes out `.mblk` and `.bbk` files.

For example, to divide the single binary mesh “`in1.msh`” into 20 pieces “`out00001.[mb]blk`”..`“out00020.[mb]blk”`, you’d use

```
$ makemblock in1.msh 20 out
```

You would then run this mesh using 20 virtual processors.

11.7 Multiblock Framework API Reference

The Multiblock framework is accessed from a program via a set of routines. These routines are available in both C and Fortran90 versions. The C versions are all functions, and always return an error code of `MBLK_SUCCESS` or `MBLK_FAILURE`. The Fortran90 versions are all subroutines, and take an extra integer parameter “`err`” which will be set to `MBLK_SUCCESS` or `MBLK_FAILURE`.

11.7.1 Initialization

All these methods should be called from the `init` function by the user. The values passed to these functions are typically read from a configuration file or computed from command-line parameters.

```
int MBLK_Set_prefix(const char *prefix);
```

```
subroutine MBLK_Set_prefix(prefix,err)
character*, intent(in)::prefix
integer, intent(out)::err
```

This function is called to set the block filename prefix. For example, if the input block files are named “`gridX00001.mblk`” and “`gridX00002.mblk`”, the prefix is the string “`gridX`”.

```
int MBLK_Set_nblocks(const int n);
```

```
subroutine MBLK_Set_nblocks(n,err)
integer, intent(in)::n
integer, intent(out)::err
```

This call is made to set the number of partitioned blocks to be used. Each block is read from an input file and a separate driver is spawned for each. The number of blocks determines the available parallelism, so be sure to have at least as many blocks as processors. We recommend using several times more blocks than processors, to ease load balancing and allow adaptive overlap of computation and communication.

Be sure to set the number of blocks equal to the number of virtual processors (+vp command-line option).

```
int MBLK_Set_dim(const int n);
```

```
subroutine MBLK_Set_dim(n, err)
integer, intent(in)::n
integer, intent(out)::err
```

This call is made to set the number of spatial dimensions. Only three dimensional computations are currently supported.

11.7.2 Utility

```
int MBLK_Get_nblocks(int* n);
```

```
subroutine MBLK_Get_nblocks(n,err)
integer, intent(out)::n
integer, intent(out)::err
```

Get the total number of blocks in the current computation. Can only be called from the driver routine.

```
int MBLK_Get_myblock(int* m);
```

```
subroutine MBLK_Get_myblock(m,err)
integer, intent(out)::m
integer, intent(out)::err
```

Get the id of the current block, an integer from 0 to the number of blocks minus one. Can only be called from the driver routine.

```
int MBLK_Get_blocksize(int* dims);
```

```
subroutine MBLK_Get_blocksize(dims,err)
integer, intent(out)::dims(3)
integer, intent(out)::err
```

Get the interior dimensions of the current block, in voxels. The size of the array dims should be 3, and will be filled with the i , j , and k dimensions of the block. Can only be called from the driver routine.

```
int MBLK_Get_nodelocs(const int* nodedim, double *nodelocs);
```

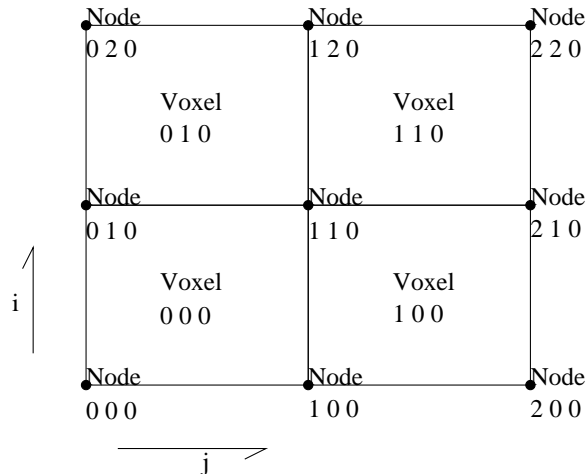


```

subroutine MBLK_Get_blocksize(nodedim,nodelocs,err)
integer,intent (in) :: nodedims(3)
double precision,intent (out) :: nodedims(3),nodedims(0),nodedims(1),nodedims(2)
integer,intent (out) :: err

```

Get the (x, y, z) locations of the nodes of the current block. The 3-array nodedim should be the number of nodes you expect, which must be exactly one more than the number of interior voxels.



21: The C node and voxel (i, j, k) numbering for a 2 x 2 voxel block. For the fortran numbering, add 1 to all indices. Ghost voxels are omitted.

You cannot obtain the locations of ghost nodes via this routine. To get the locations of ghost nodes, create a node-centered field containing the node locations and do an update field. Can only be called from the driver routine.

```

double MBLK_Timer(void);

```

```

function double precision :: MBLK_Timer()

```

Return the current wall clock time, in seconds. Resolution is machine-dependent, but is at worst 10ms.

```

void MBLK_Print_block(void);

```

```

subroutine MBLK_Print_block()

```

Print a debugging representation of the framework's information about the current block.

```

void MBLK_Print(const char *str);

```

```

subroutine MBLK_Print(str)
character*, intent (in) :: str

```

Print the given string, prepended by the block id if called from the driver. Works on all machines, unlike `printf` or `print *`, which may not work on all parallel machines.

11.7.3 Internal Boundary Conditions and Block Fields

The Multiblock framework handles the exchange of boundary values between neighboring blocks. The basic mechanism to do this exchange is the *field* – numeric data items associated with each cell of a block. These items must be

arranged in a regular 3D grid, but otherwise we make no assumptions about the meaning of a field.

You create a field once, with `MBLK_Create_Field`, then pass the resulting field ID to `MBLK_Update_Field` (which does the overlapping block communication) and/or `MBLK_Reduce_Field` (which applies a reduction over block values).

```
int MBLK_Create_Field(int *dimensions, int isVoxel, const int
base_type, const int vec_len, const int offset, const int dist, int
*fid);
```

```
subroutine MBLK_Create_Field(dimensions, isVoxel, base_type, vec_len, offset, dist,
↳err)
integer, intent(in) :: dimensions, isVoxel, base_type, vec_len, offset, dist
integer, intent(out) :: fid, err
```

Creates and returns a Multiblock field ID, which can be passed to `MBLK_Update_Field` and `MBLK_Reduce_Field`. Can only be called from driver().

`dimensions` describes the size of the array the field is in as an array of size 3, giving the i , j , and k sizes. The size should include the ghost regions – i.e., pass the actual allocated size of the array. `isVoxel` describes whether the data item is to be associated with a voxel (1, a volume-centered value) or the nodes (0, a node-centered value). `base_type` describes the type of each data item, one of:

- `MBLK_BYTE` – unsigned char, `INTEGER*1`, or `CHARACTER*1`
- `MBLK_INT` – int or `INTEGER*4`
- `MBLK_REAL` – float or `REAL*4`
- `MBLK_DOUBLE` – double, `DOUBLE PRECISION`, or `REAL*8`

`vec_len` describes the number of data items associated with each cell, an integer at least 1.

`offset` is the byte offset from the start of the array to the first interior cell's data items, a non-negative integer. This can be calculated using the `offsetof()` function, normally with `offsetof(array(1,1,1), array(interiorX, interiorY, interiorZ))`. Be sure to skip over any ghost regions.

`dist` is the byte offset from the first cell's data items to the second, a positive integer (normally the size of the data items). This can also be calculated using `offsetof()`; normally with `offsetof(array(1,1,1), array(2,1,1))`.

`fid` is the identifier for the field that is created by the function.

In the example below, we register a single double-precision value with each voxel. The ghost region is 2 cells deep along all sides.

```
!In Fortran
double precision, allocatable :: voxData(:, :, :)
integer :: size(3), ni, nj, nk
integer :: fid, err

!Find the dimensions of the grid interior
MBLK_Get_blocksize(size, err);

!Add ghost region width to the interior dimensions
size=size+4; ! 4 because of the 2-deep region on both sides

!Allocate and initialize the grid
allocate(voxData(size(1), size(2), size(3)))
voxData=0.0
```

(continues on next page)

(continued from previous page)

```
!Create a field for voxData
call MBLK_Create_field(&
    &size,1, MBLK_DOUBLE,3,&
    &offsetof(grid(1,1,1),grid(3,3,3)),&
    &offsetof(grid(1,1,1),grid(2,1,1)),fid,err)
```

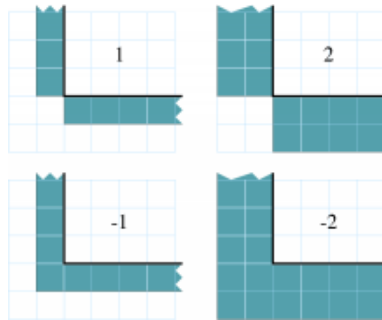
This example uses the Fortran-only helper routine `offsetof`, which returns the offset in bytes of memory between its two given variables. C users can use the built-in `sizeof` keyword or pointer arithmetic to achieve the same result.

```
void MBLK_Update_field(const int fid,int ghostwidth, void *grid);
```

```
subroutine MBLK_Update_field(fid,ghostwidth, grid,err)
integer, intent(in) :: fid, ghostwidth
integer, intent(out) :: err
varies, intent(inout) :: grid
```

Update the values in the ghost regions specified when the field was created. This call sends this block's interior region out, and receives this block's boundary region from adjoining blocks.

`ghostwidth` controls the thickness of the ghost region. To exchange only one cell on the boundary, pass 1. To exchange two cells, pass 2. To include diagonal regions, make the ghost width negative. A ghost width of zero would communicate no data.



22: The 2D ghost cells communicated for various ghost widths. The heavy line is the block interior boundary – this is the lower left portion of the block.

`MBLK_Update_field` can only be called from the driver, and to be useful, must be called from every block's driver routine.

`MBLK_Update_field` blocks until the field has been updated. After this routine returns, the given field will be updated. If the update was successful `MBLK_SUCCESS` is returned, otherwise `MBLK_FAILURE` is returned in case of error.

```
void MBLK_Iupdate_field(const int fid,int ghostwidth, void *ingrid, void* outgrid);
```

```
subroutine MBLK_Iupdate_field(fid,ghostwidth, ingrid, outgrid,err)
integer, intent(in) :: fid, ghostwidth
integer, intent(out) :: err
varies, intent(in) :: ingrid
varies, intent(out) :: outgrid
```

Update the values in the ghost regions which were specified when the field was created. For the example above the ghost regions will be updated once for each step in the time loop.

`MBLK_Iupdate_field` can only be called from the driver, and to be useful, must be called from every block's driver routine.

MBLK_Iupdate_field is a non blocking call similar to MPI_Irecv. After the routine returns the update may not yet be complete and the outgrid may be in an inconsistent state. Before using the values, the status of the update must be checked using MBLK_Test_update or MBLK_Wait_update.

There can be only one outstanding Iupdate call in progress at any time.

```
int MBLK_Test_update(int *status);
```

```
subroutine MBLK_Test_update(status,err)
integer, intent(out) :: status,err
```

MBLK_Test_update is a call that is used in association with MBLK_Iupdate_field from the driver subroutine. It tests whether the preceding Iupdate has completed or not. status is returned as MBLK_DONE if the update was completed or MBLK_NOTDONE if the update is still pending. Rather than looping if the update is still pending, call MBLK_Wait_update to relinquish the CPU.

```
void MBLK_Wait_update(void);
```

```
subroutine MBLK_Wait_update()
```

MBLK_Wait_update call is a blocking call and is used in association with MBLK_Iupdate_field call. It blocks until the update is completed.

```
void MBLK_Reduce_field(int fid,void *grid, void *out,int op);
```

```
subroutine MBLK_Reduce_field(fid,grid,outVal,op)
integer, intent(in) :: fid,op
varies, intent(in) :: grid
varies, intent(out) :: outVal
```

Combine a field from each block, according to op, across all blocks. Only the interior values of the field will be combined, not the ghost cells. After Reduce_Field returns, all blocks will have identical values in outVal, which must be vec_len copies of base_type.

May only be called from the driver, and to complete, must be called from every chunk's driver routine.

op must be one of:

- MBLK_SUM – each element of outVal will be the sum of the corresponding fields of all blocks
- MBLK_MIN – each element of outVal will be the smallest value among the corresponding field of all blocks
- MBLK_MAX – each element of outVal will be the largest value among the corresponding field of all blocks

```
void MBLK_Reduce(int fid,void *inVal,void *outVal,int op);
```

```
subroutine MBLK_Reduce(fid,inVal,outVal,op)
integer, intent(in) :: fid,op
varies, intent(in) :: inVal
varies, intent(out) :: outVal
```

Combine a field from each block, according to op, across all blocks. fid is only used for base_type and vec_len – offset and dist are not used. After this call returns, all blocks will have identical values in outVal. op has the same values and meaning as MBLK_Reduce_Field. May only be called from the driver, and to complete, must be called from every block's driver routine.

11.7.4 External Boundary Conditions

Most problems include some sort of boundary conditions. These conditions are normally applied in the ghost cells surrounding the actual computational domain. Examples of boundary conditions are imposed values, reflection walls, symmetry planes, inlets, and exits.

The Multiblock framework keeps track of where boundary conditions are to be applied. You register a subroutine that the framework will call to apply each type of external boundary condition.

```
int MBLK_Register_bc(const int bcnun, int ghostWidth, const MBLK_BcFn bcfnc);
```

```
subroutine MBLK_Register_bc(bcnun, ghostwidth, bcfnc, err)
integer,intent(in) :: bcnun, ghostWidth
integer,intent(out) :: err
subroutine :: bcfnc
```

This call is used to bind an external boundary condition subroutine, written by you, to a boundary condition number. MBLK_Register_bc should only be called from the driver.

- bcnun – The boundary condition number to be associated with the function.
- ghostWidth – The width of the ghost cells where this boundary condition is to be applied.
- bcfnc – The user subroutine to be called to apply this boundary condition.

When you ask the framework to apply boundary conditions, it will call this routine. The routine should be declared like:

```
!In Fortran
subroutine applyMyBC(param1,param2,start,end)
varies :: param1, param2
integer :: start(3), end(3)
end subroutine
```

```
/* In C */
void applyMyBC(void *param1,void *param2,int *start,int *end);
```

param1 and param2 are not used by the framework – they are passed in unmodified from MBLK_Apply_bc and MBLK_Apply_bc_all. param1 and param2 typically contain the block data and dimensions.

start and end are 3-element arrays that give the i, j, k block locations where the boundary condition is to be applied. They are both inclusive and both relative to the block interior – you must shift them over your ghost cells. The C versions are 0-based (the first index is zero), while the Fortran versions are 1-based (the first index is one).

For example, a Fortran subroutine to apply the constant value 1.0 across the boundary, with a 2-deep ghost region, would be:

```
!In Fortran
subroutine applyMyBC(grid,size,start,end)
integer :: size(3), i,j,k
double precision :: grid(size(1),size(2),size(3))
integer :: start(3), end(3)
start=start+2 ! Back up over ghost region
end=end+2
do i=start(1),end(1)
do j=start(2),end(2)
do k=start(3),end(3)
grid(i,j,k)=1.0
end do
end do
```

(continues on next page)

(continued from previous page)

```

    end do
  end do

end subroutine

```

```
int MBLK_Apply_bc(const int bcnnum, void *param1, void *param2);
```

```

subroutine MBLK_Apply_bc(bcnnum, param1,param2,err)
integer,intent(in)::bcnnum
varies,intent(inout)::param1
varies,intent(inout)::param2
integer,intent(out)::err

```

MBLK_Apply_bc call is made to apply all boundary condition functions of type `bcnnum` to the block. `param1` and `param2` are passed unmodified to the boundary condition function.

```
int MBLK_Apply_bc_all(void* param1, void* param2);
```

```

subroutine MBLK_Apply_bc_all(param1,param2, err)
integer,intent(out)::err
varies,intent(inout)::param1
varies,intent(inout)::param2

```

This call is same as MBLK_Apply_bc except it applies all external boundary conditions to the block.

11.7.5 Migration

The Charm++ runtime system includes automated, runtime load balancing, which will automatically monitor the performance of your parallel program. If needed, the load balancer can “migrate” mesh chunks from heavily-loaded processors to more lightly-loaded processors, improving the load balance and speeding up the program. For this to be useful, pass the `+vpN` argument with a larger number of blocks `N` than processors. Because this is somewhat involved, you may refrain from calling MBLK_Migrate and migration will never take place.

The runtime system can automatically move your thread stack to the new processor, but you must write a PUP function to move any global or heap-allocated data to the new processor. (Global data is declared at file scope or `static` in C and `COMMON` in Fortran77. Heap allocated data comes from C `malloc`, C++ `new`, or Fortran90 `ALLOCATE`.) A PUP (Pack/UnPack) function performs both packing (converting heap data into a message) and unpacking (converting a message back into heap data). All your global and heap data must be collected into a single block (`struct` in C, user-defined `TYPE` in Fortran) so the PUP function can access it all.

Your PUP function will be passed a pointer to your heap data block and a special handle called a “pupper”, which contains the network message to be sent. Your PUP function returns a pointer to your heap data block. In a PUP function, you pass all your heap data to routines named `pup_type`, where `type` is either a basic type (such as `int`, `char`, `float`, or `double`) or an array type (as before, but with a “s” suffix). Depending on the direction of packing, the pupper will either read from or write to the values you pass – normally, you shouldn’t even know which. The only time you need to know the direction is when you are leaving a processor or just arriving. Correspondingly, the pupper passed to you may be deleting (indicating that you are leaving the processor, and should delete your heap storage after packing), unpacking (indicating you’ve just arrived on a processor, and should allocate your heap storage before unpacking), or neither (indicating the system is merely sizing a buffer, or checkpointing your values).

PUP functions are much easier to write than explain – a simple C heap block and the corresponding PUP function is:

```

typedef struct {
    int n1; /*Length of first array below*/
    int n2; /*Length of second array below*/
    double *arr1; /*Some doubles, allocated on the heap*/
    int *arr2; /*Some ints, allocated on the heap*/
} my_block;

my_block *pup_my_block(pup_er p, my_block *m)
{
    if (pup_isUnpacking(p)) m=malloc(sizeof(my_block));
    pup_int(p, &m->n1);
    pup_int(p, &m->n2);
    if (pup_isUnpacking(p)) {
        m->arr1=malloc(m->n1*sizeof(double));
        m->arr2=malloc(m->n2*sizeof(int));
    }
    pup_doubles(p, m->arr1, m->n1);
    pup_ints(p, m->arr2, m->n2);
    if (pup_isDeleting(p)) {
        free(m->arr1);
        free(m->arr2);
        free(m);
    }
    return m;
}

```

This single PUP function can be used to copy the my_block data into a message buffer and free the old heap storage (deleting pupper), allocate storage on the new processor and copy the message data back (unpacking pupper), or save the heap data for debugging or checkpointing.

A Fortran TYPE block and corresponding PUP routine is as follows:

```

MODULE my_block_mod
  TYPE my_block
    INTEGER :: n1, n2x, n2y
    REAL*8, POINTER, DIMENSION(:) :: arr1
    INTEGER, POINTER, DIMENSION(:, :) :: arr2
  END TYPE
END MODULE

SUBROUTINE pup_my_block(p, m)
  IMPLICIT NONE
  USE my_block_mod
  USE pupmod
  INTEGER :: p
  TYPE(my_block) :: m
  call pup_int(p, m%n1)
  call pup_int(p, m%n2x)
  call pup_int(p, m%n2y)
  IF (pup_isUnpacking(p)) THEN
    ALLOCATE(m%arr1(m%n1))
    ALLOCATE(m%arr2(m%n2x, m%n2y))
  END IF
  call pup_doubles(p, m%arr1, m%n1)
  call pup_ints(p, m%arr2, m%n2x*m%n2y)
  IF (pup_isDeleting(p)) THEN
    DEALLOCATE(m%arr1)

```

(continues on next page)

(continued from previous page)

```

    DEALLOCATE (m%arr2)
  END IF
END SUBROUTINE

```

```

int MBLK_Register(void *block, MBLK_PupFn pup_ud, int* rid)

```

```

subroutine MBLK_Register(block, pup_ud, rid)
integer, intent(out) :: rid
TYPE(varies), POINTER :: block
SUBROUTINE :: pup_ud

```

Associates the given data block and PUP function. Returns a block ID, which can be passed to MBLK_Get_registered later. Can only be called from driver. It returns MBLK_SUCESS if the call was successful and MBLK_FAILURE in case of error. For the declarations above, you call MBLK_Register as:

```

/*C/C++ driver() function*/
int myId, err;
my_block *m=malloc(sizeof(my_block));
err =MBLK_Register(m, (MBLK_PupFn)pup_my_block,&rid);

```

```

!- Fortran driver subroutine
use my_block_mod
interface
  subroutine pup_my_block(p,m)
    use my_block_mod
    INTEGER :: p
    TYPE(my_block) :: m
  end subroutine
end interface
TYPE(my_block) :: m
INTEGER :: myId,err
MBLK_Register(m,pup_my_block,myId,err)

```

Note that Fortran blocks must be allocated on the stack in driver, while C/C++ blocks may be allocated on the heap.

```

void MBLK_Migrate()

```

```

subroutine MBLK_Migrate()

```

Informs the load balancing system that you are ready to be migrated, if needed. If the system decides to migrate you, the PUP function passed to MBLK_Register will be called with a sizing pupper, then a packing and deleting pupper. Your stack (and pupped data) will then be sent to the destination machine, where your PUP function will be called with an unpacking pupper. MBLK_Migrate will then return, whereupon you should call MBLK_Get_registered to get your unpacked data block. Can only be called from the driver.

```

int MBLK_Get_Userdata(int n, void** block)

```

Return your unpacked userdata after migration – that is, the return value of the unpacking call – to your PUP function. Takes the userdata ID returned by MBLK_Register. Can be called from the driver at any time.

Since Fortran blocks are always allocated on the stack, the system migrates them to the same location on the new processor, so no Get_Registered call is needed from Fortran.

Parallel Object-Oriented Simulation Environment (POSE)

Contents

- *Parallel Object-Oriented Simulation Environment (POSE)*
 - *Introduction*
 - * *Developing a model in POSE*
 - * *PDES in POSE*
 - *Compiling, Running and Debugging a Sample POSE program*
 - * *Compiling*
 - * *Running*
 - * *Debugging*
 - * *Sequential Mode*
 - *Programming in POSE*
 - * *POSE Modules*
 - * *Event Message and Poser Interface Description*
 - * *Declaring Event Messages and Posers*
 - * *Implementing Posers*
 - * *Creation of Poser Objects*
 - * *Event Method Invocations*
 - * *Elapsing Simulation Time*
 - * *Interacting with a POSE Module and the POSE System*
 - *Configuring POSE*

* *POSE Command Line Options*

- *Communication Optimizations*
- *Load Balancing*
- *Glossary of POSE-specific Terms*

12.1 Introduction

POSE (Parallel Object-oriented Simulation Environment) is a tool for parallel discrete event simulation (PDES) based on Charm++. You should have a background in object-oriented programming (preferably C++) and know the basic principles behind discrete event simulation. Familiarity with simple parallel programming in Charm++ is also a plus.

POSE uses the approach of message-driven execution of Charm++, but adds the element of discrete timestamps to control when, in simulated time, a message is executed.

Users may choose synchronization strategies (conservative or several optimistic variants) on a per class basis, depending on the desired behavior of the object. However, POSE is intended to perform best with a special type of *adaptive* synchronization strategy which changes its behavior on a per object basis. Thus, other synchronization strategies may not be properly maintained. There are two significant versions of the adaptive strategy, *adapt4*, a simple, stable version, and *adept*, the development version.

12.1.1 Developing a model in POSE

Modeling a system in POSE is similar to how you would model in C++ or any OOP language. Objects are entities that hold data, and have a fixed set of operations that can be performed on them (methods).

Charm++ provides the parallelism we desire, but the model does not differ dramatically from C++. The primary difference is that objects may exist on a set of processors, and so invoking methods on them requires communication via messages. These parallel objects are called *chares*.

POSE adds to Charm++ by putting timestamps on method invocations (events), and executing events in timestamp order to preserve the validity of the global state.

Developing a model in POSE involves identifying the entities we wish to model, determining their interactions with other entities and determining how they change over time.

12.1.2 PDES in POSE

A simulation in POSE consists of a set of Charm++ chares performing timestamped events in parallel. In POSE, these chares are called *posers*. POSE is designed to work with many such entities per processor. The more a system can be broken down into its parallel components when designing the simulation model, the more potential parallelism in the final application.

A poser class is defined with a synchronization strategy associated with it. We encourage the use of the adaptive strategies, as mentioned earlier. Adaptive strategies are optimistic, and will potentially execute events out of order, but have rollback and cancellation messages as well as checkpointing abilities to deal with this behind the scenes.

Execution is driven by events. An event arrives for a poser and is sorted into a queue by timestamp. The poser has a local time called object virtual time (OVT) which represents its progress through the simulation. When an event arrives with a timestamp $t > \text{OVT}$, the OVT is advanced to t . If the event has timestamp $t < \text{OVT}$, it may be that other events with greater timestamps were executed. If this is the case, a rollback will occur. If not, the event is simply executed along with the others in the queue.

Time can also pass on a poser within the course of executing an event. An *elapse* function is used to advance the OVT. POSE maintains a global clock, the global virtual time (GVT), that represents the progress of the entire simulation.

Currently, POSE has no way to directly specify event dependencies, so if they exist, the programmer must handle errors in ordering carefully. POSE provides a delayed error message print and abort function that is only performed if there is no chance of rolling back the dependency error. Another mechanism provided by POSE is a method of tagging events with *sequence numbers*. This allows the user to determine the execution order of events which have the same timestamp.

12.2 Compiling, Running and Debugging a Sample POSE program

Sample code is available in the Charm++ source distribution. Assuming a netlrts-linux-x86_64 build of Charm++, look in `charm/netlrts-linux-x86_64/examples/pose`. The SimBenchmark directory contains a synthetic benchmark simulation and is fairly straightforward to understand.

12.2.1 Compiling

To build a POSE simulation, run `etrans.pl` on each POSE module to get the new source files. `etrans.pl` is a source to source translator. Given a module name it will translate the `module.h`, `module.ci`, and `module.C` files into `module_sim.h`, `module_sim.ci`, and `module_sim.C` files. The translation operation adds wrapper classes for POSE objects and handles the interface with strategies and other poser options.

To facilitate code organization, the `module.C` file can be broken up into multiple files and those files can be appended to the `etrans.pl` command line after the module name. These additional `.C` files will be translated and their output appended to the `module_sim.C` file.

The `-s` switch can be passed to use the sequential simulator feature of POSE on your simulation, but you must also build a sequential version when you compile (see below).

Once the code has been translated, it is a Charm++ program that can be compiled with `charmcc`. Please refer to the CHARM++/CONVERSE Installation and Usage Manual for details on the `charmcc` command. You should build the new source files produced by `etrans.pl` along with the main program and any other source needed with `charmcc`, linking with `-module pose` (or `-module seqpose` for a sequential version) and `-language charm++`. The SimBenchmark example has a Makefile that shows this process.

12.2.2 Running

To run the program in parallel, a `charmrun` executable was created by `charmcc`. The flag `+p` is used to specify a number of processors to run the program on. For example:

```
$ ./charmrun pgm +p4
```

This runs the executable `pgm` on 4 processors. For more information on how to use `charmrun` and set up your environment for parallel runs, see the CHARM++/CONVERSE Installation and Usage Manual.

12.2.3 Debugging

Because POSE is translated to Charm++, debugging is a little more challenging than normal. Multi-processor debugging can be achieved with the `charmrun ++debug` option, and debugging is performed on the `module_sim.C` source files. The user thus has to track down problems in the original POSE source code. A long-term goal of the POSE developers is to eliminate the translation phase and rely on the interface translator of Charm++ to provide similar functionality.

12.2.4 Sequential Mode

As mentioned above, the same source code can be used to generate a purely sequential POSE executable by using the `-s` flag to `etrans.pl` and linking with `-module seqpose`. This turns off all aspects of synchronization, checkpointing and GVT calculation that are needed for optimistic parallel execution. Thus you should experience better one-processor times for executables built for sequential execution than those built for parallel execution. This is convenient for examining how a program scales in comparison to sequential time. It is also helpful for simulations that are small and fast, or in situations where multiple processors are not available.

12.3 Programming in POSE

This section details syntax and usage of POSE constructs with code samples.

12.3.1 POSE Modules

A POSE module is similar to a Charm++ module. It is comprised of an interface file with suffix `.ci`, a header `.h` file, and the implementation in `.C` files. Several posers can be described in one module, and the module can include regular chares as well. The module is translated into Charm++ before the simulation can be compiled. This translation is performed by a Perl script called `etrans.pl` which is included with POSE. It generates files suffixed `_sim.ci`, `_sim.h`, and `_sim.C`.

12.3.2 Event Message and Poser Interface Description

Messages, be they event messages or otherwise, are described in the `.ci` file exactly the way they are in Charm++. Event messages cannot make use of Charm++'s parameter marshalling, and thus you must declare them in the `.h` file. Charm++ `varsize` event messages are currently not implemented in POSE.

All event messages inherit from a POSE type `eventMsg` which includes data for timestamps and miscellaneous POSE statistics.

A message is declared in the `.ci` file as follows:

```
message myMessage;
```

Posers are described similar to chares, with a few exceptions. First, the `poser` keyword is used to denote that the class is a POSE simulation object class. Second, event methods are tagged with the keyword `event` in square brackets. Finally, three components are specified which indicate how objects of the poser class are to be simulated. The `sim` component controls the wrapper class and event queue used by the object. The `strat` component controls the synchronization strategy the object should use (*i.e.* adaptive or basic optimistic). The `rep` component specifies the global state representation, which controls how the global state is kept accurate depending on the synchronization strategy being used (*i.e.* checkpointing or no checkpointing). Currently, there is only one wrapper type, `sim`. This 3-tuple syntax is likely to become obsolete, replaced simply by synchronization strategy only. Keeping the global state accurate is largely a function of the synchronization strategy used.

```
poser mySim : sim strat rep {
  entry mySim(myMessage *); ``
  entry [event] void myEventMethod(eventMsg *);
  ...
};
```

A typical `.ci` file poser specification might look like this:

```

poser Worker : sim adapt4 chpt {
    entry Worker(WorkerCreationMsg *);
    entry [event] void doWork(WorkMsg *);
    ...
};

```

Note that the constructors and event methods of a poser must take an event message as parameter. If there is no data (and thereby no message defined) that needs to be passed to the method, then the parameter should be of type `eventMsg *`. This ensures that POSE will be able to timestamp the event.

12.3.3 Declaring Event Messages and Posers

Currently, event messages are declared with no reference to what they might inherit from (unlike in Charm++). The translator takes care of this. In addition, they must define `operator=`.

```

class myMessage {
public:
    int someData;
    myMessage& operator=(const myMessage& obj) {
        eventMsg::operator=(obj);
        someData = obj.someData;
        return *this;
    }
};

```

Similarly, posers do not refer to a base class when they are declared. Posers are required to have a void constructor declared that simply initializes the data to sensible values. A destructor must be provided as well. In addition, a `pup` and `operator=` must be provided. The `pup` method should call the `pup` method of the global state representation class being used.

```

class mySim {
    int anInt; float aFloat; char aString[20];
public:
    mySim();
    mySim(myMessage *m);
    ~mySim();
    void pup(PUP::er &p);
    mySim& operator=(const mySim& obj);
    void myEventMethod(eventMsg *m);
    void myEventMethod_anti(eventMsg *m);
    void myEventMethod_commit(eventMsg *m);
    ...
};

```

Further, for each event method, a commit method should be declared, and if the synchronization strategy being used is optimistic or involves any sort of rollback, an anti-method should also be provided. The syntax of these declarations is shown above. Their usage and implementation will be described next.

12.3.4 Implementing Posers

The void constructor for a poser should be defined however the user sees fit. It could be given an empty body and should still work for POSE. Poser entry constructors (those described in the `.ci` file) should follow the template below:

```
mySim::mySim(myMessage *m)
{
    // initializations from $m$
    ...
    delete m;
    ...
};
```

Note that while the incoming message *m* may be deleted here in the constructor, event messages received on event methods should **not** be deleted. The PDES fossil collection will take care of those.

An event method should have the following form:

```
void mySim::myEventMethod(eventMsg *m) {
    // body of method
};
```

Again, *m* is never deleted in the body of the event. A side effect of optimistic synchronization and rollback is that we would like the effects of event execution to be dependent only upon the state encapsulated in the corresponding poser. Thus, accessing arbitrary states outside of the simulation, such as by calling `rand`, is forbidden. We are planning to fix this problem by adding a `POSE_rand()` operation which will generate a random number the first time the event is executed, and will checkpoint the number for use in subsequent re-executions should a rollback occur.

12.3.5 Creation of Poser Objects

Posers are created within a module using the following syntax:

```
int hdl = 13; // handle should be unique
myMessage *m = new myMessage;
m->someData = 34;
POSE_create(mySim(m), hdl, 0);
```

This creates a `mySim` object that comes into existence at simulation time zero, and can be referred to by the handle 13.

Creating a poser from outside the module (*i.e.* from `main`) is somewhat more complex:

```
int hdl = 13;
myMessage *m = new myMessage;
m->someData = 34;
m->Timestamp(0);
(*(CProxy_mySim *) & POSE_Objects)[hdl].insert(m);
```

This is similar to what the module code ultimately gets translated to and should be replaced by a macro with similar syntax soon.

12.3.6 Event Method Invocations

Event method invocations vary significantly from entry method invocations in Charm++, and various forms should be used depending on where the event method is being invoked. In addition, event messages sent to an event method should be allocated specifically for an event invocation, and cannot be recycled or deleted.

There are three ways to send events within a POSE module. The first and most commonly used way involves specifying and offset in simulation time from the current time. The syntax follows:

```
aMsg = new eventMsg;
POSE_invoke(myEventMethod(aMsg), mySim, hdl, 0);
```

Here, we've created an `eventMsg` and sent it to `myEventMethod`, an event entry point on `mySim`. `mySim` was created at handle `hdl`, and we want the event to take place now, i.e. at the current simulation time, so the offset is zero.

The second way to send an event is reserved for use by non-poser objects within the module. It should not be used by posers. This version allows you to specify an absolute simulation time at which the event happens (as opposed to an offset to the current time). Since non-poser objects are not a part of the simulation, they do not have a current time, or OVT, by which to specify an offset. The syntax is nearly identical to that above, only the last parameter is an absolute time.

```
aMsg = new eventMsg;
POSE_invoke_at(myEventMethod(aMsg), mySim, hdl, 56);
```

Posers should not use this approach because of the risk of specifying an absolute time that is earlier than the current time on the object sending the event.

Using this method, event methods can be injected into the system from outside any module, but this is not recommended.

The third approach is useful when an object send events to itself. It is simply a slightly shorter syntax for the same thing as `POSE_invoke`:

```
aMsg = new eventMsg;
POSE_local_invoke(myEventMethod(aMsg), offset);
```

12.3.7 Elapsing Simulation Time

We've seen in the previous section how it is possible to advance simulation time by generating events with non-zero offsets of current time. When such events are received on an object, if the object is behind, it advances its local simulation time (object virtual time or OVT) to the timestamp of the event.

It is also possible to elapse time on an object while the object is executing an event. This is accomplished thus:

```
elapse(42);
```

The example above would simulate the passage of forty-two time units by adding as much to the object's current OVT.

12.3.8 Interacting with a POSE Module and the POSE System

POSE modules consist of `<modname>.ci`, `<modname>.h` and `<modname>.C` files that are translated via `etrans.pl` into `<modname>_sim.ci`, `<modname>_sim.h` and `<modname>_sim.C` files. To interface these with a main program module, say *Pgm* in files `pgm.ci`, `pgm.h` and `pgm.C`, the `pgm.ci` file must declare the POSE module as `extern` in the mainmodule `Pgm` block. For example:

```
mainmodule Pgm {
  extern module modname;
  readonly CkChareID mainhandle;

  mainchare main {
    entry main();
  };
};
```

The `pgm.C` file should include `pose.h` and `<modname>_sim.h` along with its own headers, declarations and whatever else it needs.

Somewhere in the `main` function, `POSE_init()` should be called. This initializes all of POSE's internal data structures. The parameters to `POSE_init()` specify a termination method. POSE programs can be terminated in two ways: with inactivity detection or with an end time. Inactivity detection terminates after a few iterations of the GVT if no events are being executed and virtual time is not advancing. When an end time is specified, and the GVT passes it, the simulation exits. If no parameters are provided to `POSE_init()`, then the simulation will use inactivity detection. If a time is provided as the parameter, this time will be used as the end time.

Now POSE is ready for posers. All posers can be created at this point, each with a unique handle. The programmer is responsible for choosing and keeping track of the handles created for posers. Once all posers are created, the simulation can be started:

```
POSE_start();
```

12.4 Configuring POSE

POSE can be configured in two different ways. Fundamental behaviors are controlled by altering values in the `pose_config.h` file in the POSE installation, and rebuilding POSE. Many of these configuration options can (and should) be controlled by command line options. These will be designated here by an asterisk (*). See section [12.4.1](#) for the command line options.

- `POSE_STATS_ON` *
 - Turn on timing and statistics gathering for internal POSE operations. Produces a small slowdown in program.
- `POSE_DOP_ON` *
 - Turn on timing and statistics gathering for degree of parallelism calculations. Generates log files that can be loaded by ploticus scripts to produce graphs plotting active entities over time. Slows down program dramatically.
- `POSE_COMM_ON`
 - Turn on streaming communication optimization for small message packing.
- `COMM_TIMEOUT`
 - Used by streaming communication library. Time to wait (in ?) before sending buffered messages.
- `COMM_MAXMSG`
 - Used by streaming communication library. Number of messages to buffer before packing and sending as one.
- `LB_ON` *
 - Turn on POSE load balancing.
- `STORE_RATE` *
 - Default checkpointing rate: 1 for every `STORE_RATE` events.
- `SPEC_WINDOW` *
 - Speculative window size: this is how far (in virtual time units) ahead of GVT posers are allowed to go.
- `MIN_LEASH` * and `MAX_LEASH` *
 - Bounds on the speculative window, these are adjusted by adaptive synchronization strategies.
- `LEASH_FLEX` *
 - Granularity of flexibility when speculative window is shrunk or expanded.
- `MAX_POOL_SIZE`

- Memory used by event messages is recycled. This controls how many messages of a particular size will be kept on hand.
- `MAX_RECYCLABLE`
 - This is the largest size of message that will be recycled.
- `LB_SKIP` *
 - This controls the frequency of load balance invocation. 1 in every `LB_SKIP` executions of the GVT algorithm will invoke load balancing.
- `LB_THRESHOLD` *
 - What the heck does this number mean? I can't remember. I'll have to look through the code... later. Meanwhile, I think this indicates some sort of threshold a single processor has to cross before we even bother with analyzing the load.
- `LB_DIFF` *
 - Once the load has been analyzed, we compute the difference between the max and min PE loads. Only if this difference exceeds `LB_DIFF` do we bother migrating posers.

Several of the above flags and constants will be eliminated as the adaptive strategy is expanded. What remains will eventually become run-time options.

12.4.1 POSE Command Line Options

Command line options are handled like Charm++ command line parameters. For namespace purity all POSE command line options have a `_pose` suffix. They can be inspected by appending a `-h` to an execution of a POSE program. Command line options override any defaults set in the `pose_config.h` file

- `+stats_pose`
 - Turn on timing and statistics gathering for internal POSE operations. Produces a small slowdown in program.
- `+dop_pose`
 - Turn on timing and statistics gathering for degree of parallelism calculations. Generates log files that can be loaded by ploticus scripts to produce graphs plotting active entities over time. Slows down program dramatically.
- `+lb_on_pose`
 - Turn on POSE load balancing.
- `+store_rate_pose N`
 - Default checkpointing rate: 1 for every `STORE_RATE` events.
- `+spec_window_pose N`
 - Speculative window size: this is how far (in virtual time units) ahead of GVT posers are allowed to go.
- `+min_leash_pose N` and `+min_leash_pose N`
 - Bounds on the speculative window, these are adjusted by adaptive synchronization strategies.
- `+leash_flex_pose N`
 - Granularity of flexibility when speculative window is shrunk or expanded.
- `+lb_skip_pose N`
 - This controls the frequency of load balance invocation. 1 in every `LB_SKIP` executions of the GVT algorithm will invoke load balancing.
- `+lb_threshold_pose N`
 - Minimum threshold for load balancing, default is 4000
- `+lb_diff_pose N`

- Once the load has been analyzed, we compute the difference between the max and min PE loads. Only if this difference exceeds `LB_DIFF` do we bother migrating posers.
- `+checkpoint_gvt_pose N`
 - Checkpoint to disk approximately every `N` GVT ticks (`N` is an integer). The default is 0, which indicates no checkpointing.
- `+checkpoint_time_pose N`
 - Checkpoint to disk every `N` seconds (`N` is an integer). The default is 0, which indicates no checkpointing. If both this parameter and `+checkpoint_gvt_pose` are greater than 0, a warning will be given, the value of this parameter will be set to 0, and POSE will checkpoint based on GVT ticks.

As a technical point, pose command line parsing is done inside the `POSE_init()` call. Therefore, the most consistent behavior for interleaving pose command line options with user application options will be achieved by calling `POSE_init()` before handling user application command line arguments.

12.5 Communication Optimizations

12.6 Load Balancing

12.7 Glossary of POSE-specific Terms

- `void POSE_init()`
 - Initializes various items in POSE; creates the load balancer if load balancing is turned on; initializes the statistics gathering facility if statistics are turned on.
 - Must be called in user's main program prior to creation of any simulation objects or reference to any other POSE construct.
- `void POSE_start()`
 - Sets busy wait to default if none specified; starts quiescence detection; starts simulation timer.
 - Must be called in user's main program when simulation should start.
- `void POSE_registerCallBack(CkCallback cb)`
 - Registers callback function with POSE - when program ends or quiesces, function is called.
 - `CkCallback` is created with the index of the callback function and a proxy to the object that function is to be called on. For example, to register the function `wrapUp` in the main module as a callback:

```
CProxy_main M(mainhandle);  
POSE_registerCallBack(CkCallback(CkIndex_main::wrapUp(), M));
```

- `void POSE_stop()`
 - Commits remaining events; prints final time and statistics (if on); calls callback function.
 - Called internally when quiescence is detected or program reaches `POSE_endtime`.
- `void POSE_exit()`
 - Similar to `CkExit()`.
- `void POSE_set_busy_wait(int n)`
 - Used to control granularity of events; when calling `POSE_busy_wait`, program busywaits for time to compute *fib*(*n*).
- `void POSE_busy_wait()`
 - Busywait for time to compute *fib*(*n*) where *n* is either 1 or set by `POSE_set_busy_wait`.

- `POSE_useET(t)`
 - Set program to terminate when global virtual time (GVT) reaches *t*.
- `POSE_useID()`
 - Set program to terminate when no events are available in the simulation.
- `void POSE_create(constructorName(eventMsg *m), int handle, int atTime)`
 - Creates a poser object given its constructor, an event message *m* of the appropriate type, any integer as the handle (by which the object will be referred from then on), and a time (in simulation timesteps) at which it should be created.
 - The handle can be thought of as a chare array element index in Charm++.
- `void POSE_invoke_at(methodName(eventMsg *m), className, int handle, int atTime)`
 - Send a *methodName* event with message *m* to an object of type *className* designated by handle *handle* at time specified by *atTime*.
 - This can be used by non-poser objects in the POSE module to inject events into the system being simulated. It should not be used by a poser object to generate an event.
- `void POSE_invoke(methodName(eventMsg *m), className, int handle, int timeOffset)`
 - Send a *methodName* event with message *m* to an object of type *className* designated by handle *handle* at current OVT + *timeOffset*.
 - This is used by poser objects to send events from one poser to another.
- `void POSE_local_invoke(methodName(eventMsg *m), int timeOffset)`
 - Send a *methodName* event with message *m* to this object at current OVT + *timeOffset*.
 - This is used by poser objects to send events to themselves.
- `void CommitPrintf(char *s, args...)`
 - Buffered print statement; prints when event is committed (i.e. will not be rolled back).
 - Currently, must be called on the wrapper class (parent) to work properly, but a fix for this is in the works.
- `void CommitError(char *s, args...)`
 - Buffered error statement; prints and aborts program when event is committed.
 - Currently, must be called on the wrapper class (parent) to work properly, but a fix for this is in the works.
- `void elapse(int n)`
 - Elapse *n* simulation time units.
- `poser`
 - Keyword (used in place of chare) to denote a poser object in the `.ci` file of a POSE module.
- `event`
 - Keyword used in square brackets in the `.ci` file of a POSE module to denote that the entry method is an event method.
- `eventMsg`
 - Base class for all event messages; provides timestamp, priority and many other properties.
- `sim`
 - Base class of all wrapper classes.
- `strat`
 - Base class of all strategy classes.
- `con`
 - Simple conservative strategy class.

- `opt`, `opt2`, `opt3`, `spec`, `adapt`, `adapt2`
 - Optimistic strategy classes.
- `rep`
 - Base class for all representation classes.
- `chpt`
 - Simple checkpointing representation class.
- `OVT()`
 - Returns the object virtual time (OVT) of the posers in which it is called
- `void MySim::terminus()`
 - When simulation has terminated and program is about to exit, this method is called on all posers. Implemented as an empty method in the base `rep` class, the programmer may choose to override this with whatever actions may need to be performed per object at the end of the simulation.

Converse Programming

Contents

- *Converse Programming*
 - *Initialization and Completion*
 - *Machine Interface and Scheduler*
 - * *Machine Model*
 - * *Defining Handler Numbers*
 - * *Writing Handler Functions*
 - * *Building Messages*
 - * *Sending Messages*
 - * *Broadcasting Messages*
 - * *Multicasting Messages*
 - * *Reducing Messaging*
 - * *Scheduling Messages*
 - * *Polling for Messages*
 - * *The Timer*
 - * *Processor Ids*
 - * *Global Variables and Utility functions*
 - * *Input/Output*
 - * *Spanning Tree Calls*
 - * *Isomalloc*

- *Threads*
 - * *Basic Thread Calls*
 - * *Thread Scheduling and Blocking Restrictions*
 - * *Thread Scheduling Hooks*
- *Timers, Periodic Checks, and Conditions*
- *Converse Client-Server Interface*
 - * *CCS: Starting a Server*
 - * *CCS: Client-Side*
 - * *CCS: Server-Side*
 - * *CCS: system handlers*
 - * *CCS: network protocol*
 - * *CCS: Authentication*
- *Converse One Sided Communication Interface*
 - * *Registering / Unregistering Memory for RDMA*
 - * *RDMA operations (Get / Put)*
 - * *Completion of RDMA operation*
- *Random Number Generation*
 - * *Default Stream Calls*
 - * *Private Stream Calls*
- *Converse Persistent Communication Interface*
 - * *Create / Destroy Persistent Handler*
 - * *Persistent Operation*

13.1 Initialization and Completion

The program utilizing Converse begins executing at `main`, like any other C program. The initialization process is somewhat complicated by the fact that hardware vendors don't agree about which processors should execute `main`. On some machines, every processor executes `main`. On others, only one processor executes `main`. All processors which don't execute `main` are asleep when the program begins. The function `ConverseInit` is used to start the Converse system, and to wake up the sleeping processors.

```
typedef void (*CmiStartFn) (int argc, char **argv);  
void ConverseInit(int argc, char *argv[], CmiStartFn fn, int usched, int  
initret);
```

This function starts up the Converse system. It can execute in one of the modes described below.

Normal Mode: `schedmode=0, initret=0`

When the user runs a program, some of the processors automatically invoke `main`, while others remain asleep. All processors which automatically invoked `main` must call `ConverseInit`. This initializes the entire Converse system. Converse then initiates, on *all* processors, the execution of the user-supplied start-function `fn(argc, argv)`.

When this function returns, Converse automatically calls `CsdScheduler`, a function that polls for messages and executes their handlers (see chapter 2). Once `CsdScheduler` exits on all processors, the Converse system shuts down, and the user's program terminates. Note that in this case, `ConverseInit` never returns. The user is not allowed to poll for messages manually.

User-calls-scheduler Mode: `schedmode=1, initret=0`

If the user wants to poll for messages and other events manually, this mode is used to initialize Converse. In normal mode, it is assumed that the user-supplied start-function `fn(argc, argv)` is just for initialization, and that the remainder of the lifespan of the program is spent in the (automatically-invoked) function `CsdScheduler`, polling for messages. In user-calls-scheduler mode, however, it is assumed that the user-supplied start-function will perform the *entire computation*, including polling for messages. Thus, `ConverseInit` will not automatically call `CsdScheduler` for you. When the user-supplied start-function ends, Converse shuts down. This mode is not supported on the sim version. This mode can be combined with `ConverseInit-returns` mode below.

ConverseInit-returns Mode: `schedmode=1, initret=1`

This option is used when you want `ConverseInit` to return. All processors which automatically invoked `main` must call `ConverseInit`. This initializes the entire Converse System. On all processors which *did not* automatically invoke `main`, Converse initiates the user-supplied initialization function `fn(argc, argv)`. Meanwhile, on those processors which *did* automatically invoke `main`, `ConverseInit` returns. Shutdown is initiated when the processors that *did* automatically invoke `main` call `ConverseExit`, and when the other processors return from `fn`. The optional exit code is returned to the calling shell. If no value is specified, an exit code of zero is returned. In this mode, all polling for messages must be done manually (probably using `CsdScheduler` explicitly). This option is not supported by the sim version.

```
void ConverseExit(int exitcode /*optional*/)
```

This function is only used in `ConverseInit-returns` mode, described above.

```
void CmiAbort(char *msg, ...)
```

This function can be used portably to abnormally terminate a Converse program. Before termination, it prints a message supplied as `msg`.

```
void CmiAssert(int expr)
```

This macro terminates the Converse program after printing an informative message if `expr` evaluates to 0. It can be used in place of `assert`. In order to turn off `CmiAssert`, one should define `CMK_OPTIMIZE` as 1.

13.2 Machine Interface and Scheduler

This chapter describes two of Converse's modules: the CMI, and the CSD. Together, they serve to transmit messages and schedule the delivery of messages. First, we describe the machine model assumed by Converse.

13.2.1 Machine Model

Converse treats the parallel machine as a collection of *nodes*, where each node is comprised of a number of *processors* that share memory. In some cases, the number of processors per node may be exactly one (e.g. Distributed memory multicomputers such as IBM SP.) In addition, each of the processors may have multiple *threads* running on them which share code and data but have different stacks. Functions and macros are provided for handling shared memory across processors and querying node information. These are discussed in Section 13.2.13.

13.2.2 Defining Handler Numbers

When a message arrives at a processor, it triggers the execution of a *handler function*, not unlike a UNIX signal handler. The handler function receives, as an argument, a pointer to the message. The message itself specifies which handler function is to be called when the message arrives. Messages are contiguous sequences of bytes. The message has two parts: the header, and the data. The data may contain anything you like. The header contains a *handler number*, which specifies which handler function is to be executed when the message arrives. Before you can send a message, you have to define the handler numbers.

Converse maintains a table mapping handler numbers to function pointers. Each processor has its own copy of the mapping. There is a caution associated with this approach: it is the user's responsibility to ensure that all processors have identical mappings. This is easy to do, nonetheless, and the user must be aware that this is (usually) required.

The following functions are provided to define the handler numbers:

```
typedef void (*CmiHandler) (void *)
```

Functions that handle Converse messages must be of this type.

```
int CmiRegisterHandler(CmiHandler h)
```

This represents the standard technique for associating numbers with functions. To use this technique, the Converse user registers each of his functions, one by one, using `CmiRegisterHandler`. One must register exactly the same functions in exactly the same order on all processors. The system assigns monotonically increasing numbers to the functions, the same numbers on all processors. This insures global consistency. `CmiRegisterHandler` returns the number which was chosen for the function being registered.

```
int CmiRegisterHandlerGlobal(CmiHandler h)
```

This represents a second registration technique. The Converse user registers his functions on processor zero, using `CmiRegisterHandlerGlobal`. The Converse user is then responsible for broadcasting those handler numbers to other processors, and installing them using `CmiNumberHandler` below. The user should take care not to invoke those handlers until they are fully installed.

```
int CmiRegisterHandlerLocal(CmiHandler h)
```

This function is used when one wishes to register functions in a manner that is not consistent across processors. This function chooses a locally-meaningful number for the function, and records it locally. No attempt is made to ensure consistency across processors.

```
void CmiNumberHandler(int n, CmiHandler h)
```

Forces the system to associate the specified handler number `n` with the specified handler function `h`. If the function number `n` was previously mapped to some other function, that old mapping is forgotten. The mapping that this function creates is local to the current processor. `CmiNumberHandler` can be useful in combination with `CmiRegisterGlobalHandler`. It can also be used to implement user-defined numbering schemes: such schemes should keep in mind that the size of the table that holds the mapping is proportional to the largest handler number — do not use big numbers!

(**Note:** Of the three registration methods, the `CmiRegisterHandler` method is by far the simplest, and is strongly encouraged. The others are primarily to ease the porting of systems that already use similar registration techniques. One may use all three registration methods in a program. The system guarantees that no numbering conflicts will occur as a result of this combination.)

13.2.3 Writing Handler Functions

A message handler function is just a C function that accepts a void pointer (to a message buffer) as an argument, and returns nothing. The handler may use the message buffer for any purpose, but is responsible for eventually deleting the message using `CmiFree`.

13.2.4 Building Messages

To send a message, one first creates a buffer to hold the message. The buffer must be large enough to hold the header and the data. The buffer can be in any kind of memory: it could be a local variable, it could be a global, it could be allocated with `malloc`, and finally, it could be allocated with `CmiAlloc`. The Converse user fills the buffer with the message data. One puts a handler number in the message, thereby specifying which handler function the message should trigger when it arrives. Finally, one uses a message-transmission function to send the message.

The following functions are provided to help build message buffers:

```
void *CmiAlloc(int size)
```

Allocates memory of size `size` in heap and returns pointer to the usable space. There are some message-sending functions that accept only message buffers that were allocated with `CmiAlloc`. Thus, this is the preferred way to allocate message buffers. The returned pointer point to the message header, the user data will follow it. See `CmiMsgHeaderSizeBytes` for this.

```
void CmiFree(void *ptr)
```

This function frees the memory pointed to by `ptr`. `ptr` should be a pointer that was previously returned by `CmiAlloc`.

```
#define CmiMsgHeaderSizeBytes
```

This constant contains the size of the message header. When one allocates a message buffer, one must set aside enough space for the header and the data. This macro helps you to do so. For example, if one want to allocate an array of 100 `int`, he should call the function this way:

```
void CmiSetHandler(int *MessageBuffer, int HandlerId)
```

This macro sets the handler number of a message to `HandlerId`.

```
int CmiGetHandler(int *MessageBuffer)
```

This call returns the handler of a message in the form of a handler number.

```
CmiHandler CmiGetHandlerFunction(int *MessageBuffer)
```

This call returns the handler of a message in the form of a function pointer.

13.2.5 Sending Messages

The following functions allow you to send messages. Our model is that the data starts out in the message buffer, and from there gets transferred “into the network”. The data stays “in the network” for a while, and eventually appears on the target processor. Using that model, each of these send-functions is a device that transfers data into the network. None of these functions wait for the data to be delivered.

On some machines, the network accepts data rather slowly. We don’t want the process to sit idle, waiting for the network to accept the data. So, we provide several variations on each send function:

- **sync**: a version that is as simple as possible, pushing the data into the network and not returning until the data is “in the network”. As soon as a sync function returns, you can reuse the message buffer.
- **async**: a version that returns almost instantaneously, and then continues working in the background. The background job transfers the data from the message buffer into the network. Since the background job is still using the message buffer when the function returns, you can’t reuse the message buffer immediately. The background job sets a flag when it is done and you can then reuse the message buffer.
- **send and free**: a version that returns almost instantaneously, and then continues working in the background. The background job transfers the data from the message buffer into the network. When the background job finishes, it CmiFrees the message buffer. In this situation, you can’t reuse the message buffer at all. To use a function of this type, you must allocate the message buffer using CmiAlloc.
- **node**: a version that send a message to a node instead of a specific processor. This means that when the message is received, any “free” processor within than node can handle it.

```
void CmiSyncSend(unsigned int destPE, unsigned int size, void *msg)
```

Sends msg of size size bytes to processor destPE. When it returns, you may reuse the message buffer.

```
void CmiSyncNodeSend(unsigned int destNode, unsigned int size, void *msg)
```

Sends msg of size size bytes to node destNode. When it returns, you may reuse the message buffer.

```
void CmiSyncSendAndFree(unsigned int destPE, unsigned int size, void *msg)
```

Sends msg of size size bytes to processor destPE. When it returns, the message buffer has been freed using CmiFree.

```
void CmiSyncNodeSendAndFree(unsigned int destNode, unsigned int size, void *msg)
```

Sends msg of size size bytes to node destNode. When it returns, the message buffer has been freed using CmiFree.

```
CmiCommHandle CmiAsyncSend(unsigned int destPE, unsigned int size, void *msg)
```

Sends msg of size size bytes to processor destPE. It returns a communication handle which can be tested using CmiAsyncMsgSent: when this returns true, you may reuse the message buffer. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent.

```
CmiCommHandle CmiAsyncNodeSend(unsigned int destNode, unsigned int size, void *msg)
```

Sends msg of size size bytes to node destNode. It returns a communication handle which can be tested using CmiAsyncMsgSent: when this returns true, you may reuse the message buffer. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent.

```
void CmiSyncVectorSend(int destPE, int len, int sizes[], char *msgComps[])
```

Concatenates several pieces of data and sends them to processor destPE. The data consists of len pieces residing in different areas of memory, which are logically concatenated. The msgComps array contains pointers to the pieces; the size of msgComps[i] is taken from sizes[i]. When it returns, sizes, msgComps and the message components specified in msgComps can be immediately reused.

```
void CmiSyncVectorSendAndFree(int destPE, int len, int sizes[], char *msgComps[])
```

Concatenates several pieces of data and sends them to processor destPE. The data consists of len pieces residing in different areas of memory, which are logically concatenated. The msgComps array contains pointers to the pieces; the size of msgComps[i] is taken from sizes[i]. The message components specified in msgComps are CmiFreed by this function therefore, they should be dynamically allocated using CmiAlloc. However, the sizes and msgComps array themselves are not freed.

```
CmiCommHandle CmiAsyncVectorSend(int destPE, int len, int sizes[], char *msgComps[])
```

Concatenates several pieces of data and sends them to processor destPE. The data consists of len pieces residing in different areas of memory, which are logically concatenated. The msgComps array contains pointers to the pieces; the size of msgComps[i] is taken from sizes[i]. The individual pieces of data as well as the arrays sizes and msgComps should not be overwritten or freed before the communication is complete. This function returns a communication handle which can be tested using CmiAsyncMsgSent: when this returns true, the input parameters can be reused. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent.

```
int CmiAsyncMsgSent(CmiCommHandle handle)
```

Returns true if the communication specified by the given CmiCommHandle has proceeded to the point where the message buffer can be reused.

```
void CmiReleaseCommHandle(CmiCommHandle handle)
```

Releases the communication handle handle and associated resources. It does not free the message buffer.

```
void CmiMultipleSend(unsigned int destPE, int len, int sizes[], char *msgComps[])
```

This function allows the user to send multiple messages that may be destined for the SAME PE in one go. This is more efficient than sending each message to the destination node separately. This function assumes that the handlers that are to receive this message have already been set. If this is not done, the behavior of the function is undefined.

In the function, The destPE parameter identifies the destination processor. The len argument identifies the *number* of messages that are to be sent in one go. The sizes[] array is an array of sizes of each of these messages. The msgComps[] array is the array of the messages. The indexing in each array is from 0 to len - 1. (**Note:** Before calling this function, the program needs to initialize the system to be able to provide this service. This is done by calling the function CmiInitMultipleSendRoutine. Unless this function is called, the system will not be able to provide the service to the user.)

13.2.6 Broadcasting Messages

```
void CmiSyncBroadcast(unsigned int size, void *msg)
```

Sends msg of length size bytes to all processors excluding the processor on which the caller resides.

```
void CmiSyncNodeBroadcast(unsigned int size, void *msg)
```

Sends msg of length size bytes to all nodes excluding the node on which the caller resides.

```
void CmiSyncBroadcastAndFree(unsigned int size, void *msg)
```

Sends msg of length size bytes to all processors excluding the processor on which the caller resides. Uses CmiFree to deallocate the message buffer for msg when the broadcast completes. Therefore msg must point to a buffer allocated with CmiAlloc.

```
void CmiSyncNodeBroadcastAndFree(unsigned int size, void *msg)
```

Sends msg of length size bytes to all nodes excluding the node on which the caller resides. Uses CmiFree to deallocate the message buffer for msg when the broadcast completes. Therefore msg must point to a buffer allocated with CmiAlloc.

```
void CmiSyncBroadcastAll(unsigned int size, void *msg)
```

Sends msg of length size bytes to all processors including the processor on which the caller resides. This function does not free the message buffer for msg.

```
void CmiSyncNodeBroadcastAll(unsigned int size, void *msg)
```

Sends msg of length size bytes to all nodes including the node on which the caller resides. This function does not free the message buffer for msg.

```
void CmiSyncBroadcastAllAndFree(unsigned int size, void *msg)
```

Sends msg of length size bytes to all processors including the processor on which the caller resides. This function frees the message buffer for msg before returning, so msg must point to a dynamically allocated buffer.

```
void CmiSyncNodeBroadcastAllAndFree(unsigned int size, void *msg)
```

Sends msg of length size bytes to all nodes including the node on which the caller resides. This function frees the message buffer for msg before returning, so msg must point to a dynamically allocated buffer.

```
CmiCommHandle CmiAsyncBroadcast(unsigned int size, void *msg)
```

Initiates asynchronous broadcast of message msg of length size bytes to all processors excluding the processor on which the caller resides. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. msg should not be overwritten or freed before the communication is complete.

```
CmiCommHandle CmiAsyncNodeBroadcast(unsigned int size, void *msg)
```

Initiates asynchronous broadcast of message msg of length size bytes to all nodes excluding the node on which the caller resides. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. msg should not be overwritten or freed before the communication is complete.

```
CmiCommHandle CmiAsyncBroadcastAll(unsigned int size, void *msg)
```

Initiates asynchronous broadcast of message msg of length size bytes to all processors including the processor on which the caller resides. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. msg should not be overwritten or freed before the communication is complete.

```
CmiCommHandle CmiAsyncNodeBroadcastAll(unsigned int size, void *msg)
```

Initiates asynchronous broadcast of message msg of length size bytes to all nodes including the node on which the caller resides. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. msg should not be overwritten or freed before the communication is complete.

13.2.7 Multicasting Messages

```
typedef ... CmiGroup;
```

A CmiGroup represents a set of processors. It is an opaque type. Group IDs are useful for the multicast functions below.

```
CmiGroup CmiEstablishGroup(int npes, int *pes);
```

Converts an array of processor numbers into a group ID. Group IDs are useful for the multicast functions below. Caution: this call uses up some resources. In particular, establishing a group uses some network bandwidth (one broadcast's worth) and a small amount of memory on all processors.

```
void CmiSyncMulticast(CmiGroup grp, unsigned int size, void *msg)
```

Sends msg of length size bytes to all members of the specified group. Group IDs are created using CmiEstablishGroup.

```
void CmiSyncMulticastAndFree(CmiGroup grp, unsigned int size, void *msg)
```

Sends msg of length size bytes to all members of the specified group. Uses CmiFree to deallocate the message buffer for msg when the broadcast completes. Therefore msg must point to a buffer allocated with CmiAlloc. Group IDs are created using CmiEstablishGroup.

```
CmiCommHandle CmiAsyncMulticast(CmiGroup grp, unsigned int size, void *msg)
```

(Note: Not yet implemented.) Initiates asynchronous broadcast of message msg of length size bytes to all members of the specified group. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. msg should not be overwritten or freed before the communication is complete. Group IDs are created using CmiEstablishGroup.

```
void CmiSyncListSend(int npes, int *pes, unsigned int size, void *msg)
```

Sends msg of length size bytes to npes processors in the array pes.

```
void CmiSyncListSendAndFree(int npes, int *pes, unsigned int size, void *msg)
```

Sends msg of length size bytes to npes processors in the array pes. Uses CmiFree to deallocate the message buffer for msg when the multicast completes. Therefore, msg must point to a buffer allocated with CmiAlloc.

```
CmiCommHandle CmiAsyncListSend(int npes, int *pes, unsigned int size, void *msg)
```

Initiates asynchronous multicast of message msg of length size bytes to npes processors in the array pes. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. msg should not be overwritten or freed before the communication is complete.

13.2.8 Reducing Messaging

Reductions are operations for which a message (or user data structure) is contributed by each participant processor. All these contributions are merged according to a merge-function provided by the user. A Converse handler is then invoked with the resulting message. Reductions can be on the entire set of processors, or on a subset of the whole.

There are eight functions used to deposit a message into the system, summarized in Table 10. Half of them receive as contribution a Converse message (with a Converse header at its beginning). This message must have already been set for delivery to the desired handler. The other half (ending with "Struct") receives a pointer to a data structure allocated by the user. This second version may allow the user to write a simpler merging function. For instance, the data structure could be a tree that can be easily expanded by adding more nodes.

10: Reductions functions in Converse

	global	global with ID	processor set	CmiGroup
message	CmiReduce	CmiReduceID	CmiListReduce	CmiGroupReduce
data	CmiReduceStruct	CmiReduceStructID	CmiListReduceStruct	CmiGroupReduceStruct

The signatures for the functions in Table 10 are:

```

void CmiReduce(void *msg, int size, CmiReduceMergeFn mergeFn);

void CmiReduceStruct(void *data, CmiReducePupFn pupFn,
CmiReduceMergeFn mergeFn, CmiHandler dest, CmiReduceDeleteFn deleteFn);

void CmiReduceID(void *msg, int size, CmiReduceMergeFn mergeFn,
CmiReductionID id);

void CmiReduceStructID(void *data, CmiReducePupFn pupFn, CmiReduceMergeFn mergeFn,
CmiHandler dest, CmiReduceDeleteFn deleteFn, CmiReductionID id);

void CmiListReduce(int npes, int *pes, void *msg, int size,
CmiReduceMergeFn mergeFn, CmiReductionID id);

void CmiListReduceStruct(int npes, int *pes, void *data, CmiReducePupFn
pupFn, CmiReduceMergeFn mergeFn, CmiHandler dest, CmiReduceDeleteFn
deleteFn, CmiReductionID id);

void CmiGroupReduce(CmiGroup grp, void *msg, int size,
CmiReduceMergeFn mergeFn, CmiReductionID id);

void CmiGroupReduceStruct(CmiGroup grp, void *data, CmiReducePupFn pupFn,
CmiReduceMergeFn mergeFn, CmiHandler dest, CmiReduceDeleteFn deleteFn,
CmiReductionID id);

```

Additionally, there are variations of the global reduction functions that operate on a per-node basis, instead of per-PE.

11: Node reductions functions in Converse

	global	global with ID
message	CmiNodeReduce	CmiNodeReduceID
data	CmiNodeReduceStruct	CmiNodeReduceStructID

The signatures for the functions in Table 11 are:

```

void CmiNodeReduce(void *msg, int size, CmiReduceMergeFn mergeFn);

void CmiNodeReduceStruct(void *data, CmiReducePupFn pupFn,
CmiReduceMergeFn mergeFn, CmiHandler dest, CmiReduceDeleteFn deleteFn);

void CmiNodeReduceID(void *msg, int size, CmiReduceMergeFn mergeFn,
CmiReductionID id);

void CmiNodeReduceStructID(void *data, CmiReducePupFn pupFn, CmiReduceMergeFn mergeFn,
CmiHandler dest, CmiReduceDeleteFn deleteFn, CmiReductionID id);

```

In all the above, msg is the Converse message deposited by the local processor, size is the size of the message msg, and data is a pointer to the user-allocated data structure deposited by the local processor. dest is the CmiHandler where

the final message shall be delivered. It is explicitly passed in “Struct” functions only, since for the message versions it is taken from the header of msg. Moreover there are several other function pointers passed in by the user:

```
void * (*mergeFn) (int *size, void *local, void **remote, int count)
```

Prototype for a CmiReduceMergeFn function pointer argument. This function is used in all the CmiReduce forms to merge the local message/data structure deposited on a processor with all the messages incoming from the children processors of the reduction spanning tree. The input parameters are in the order: the size of the local data for message reductions (always zero for struct reductions); the local data itself (the exact same pointer passed in as first parameter of CmiReduce and similar); a pointer to an array of incoming messages; the number of elements in the second parameter. The function returns a pointer to a freshly allocated message (or data structure for the Struct forms) corresponding to the merge of all the messages. When performing message reductions, this function is also responsible to updating the integer pointed by size to the new size of the returned message. All the messages in the remote array are deleted by the system; the data pointed by the first parameter should be deleted by this function. If the data can be merged “in-place” by modifying or augmenting local, the function can return the same pointer to local which can be considered freshly allocated. Each element in remote is the complete incoming message (including the converse header) for message reductions, and the data as it has been packed by the pup function (without any additional header) for struct reductions.

```
void (*pupFn) (pup_er p, void *data)
```

Prototype for a CmiReducePupFn function pointer argument. This function will use the PUP framework to pup the data passed in into a message for sending across the network. The data can be either the same data passed in as first parameter of any “Struct” function, or the return of the merge function. It will be called for sizing and packing. (Note: It will not be called for unpacking.)

```
void (*deleteFn) (void *ptr)
```

Prototype for a CmiReduceDeleteFn function pointer argument. This function is used to delete either the data structure passed in as first parameter of any “Struct” function, or the return of the merge function. It can be as simple as “free” or as complicated as needed to delete complex structures. If this function is NULL, the data structure will not be deleted, and the program can continue to use it. Note: even if this function is NULL, the input data structure may still be modified by the merge function.

CmiReduce and CmiReduceStruct are the simplest reduction function, and they reduce the deposited message/data across all the processors in the system. Each processor must to call this function exactly once. Multiple reductions can be invoked without waiting for previous ones to finish, but the user is responsible to call CmiReduce/CmiReduceStruct in the same order on every processor. (**Note:** CmiReduce and CmiReduceStruct are not interchangeable. Either every processor calls CmiReduce or every processor calls CmiReduceStruct).

In situations where it is not possible to guarantee the order of reductions, the user may use CmiReduceID or CmiReduceStructID. These functions have an additional parameter of type CmiReductionID which will uniquely identify the reduction, and match them correctly. (**Note:** No two reductions can be active at the same time with the same CmiReductionID. It is up to the user to guarantee this.)

CmiNodeReduce, CmiNodeReduceStruct, CmiNodeReduceID, and CmiNodeReduceStructID are the same, but for nodes instead of PEs.

A CmiReductionID can be obtained by the user in three ways, using one of the following functions:

```
CmiReductionID CmiGetGlobalReduction()
```

This function must be called on every processor, and in the same order if called multiple times. This would generally be inside initialization code, that can set aside some CmiReductionIDs for later use.

```
CmiReductionID CmiGetDynamicReduction()
```


This function may be called only on processor zero. It returns a unique ID, and it is up to the user to distribute this ID to any processor that needs it.

```
void CmiGetDynamicReductionRemote(int handlerIdx, int pe, int dataSize, void *data)
```

This function may be called on any processor. The produced CmiReductionID is returned on the specified pe by sending a message to the specified handlerIdx. If pe is -1, then all processors will receive the notification message. data can be any data structure that the user wants to receive on the specified handler (for example to differentiate between requests). dataSize is the size in bytes of data. If dataSize is zero, data is ignored. The message received by handlerIdx consists of the standard Converse header, followed by the requested CmiReductionID (represented as a 4 bytes integer the user can cast to a CmiReductionID, a 4 byte integer containing dataSize, and the data itself.

```
CmiReductionID CmiGetGlobalNodeReduction()
CmiReductionID CmiGetDynamicNodeReduction()
void CmiGetDynamicNodeReductionRemote(int handlerIdx, int node, int dataSize, void _
↪ *data)
```

Same as above, but for nodes instead of PEs.

The other four functions (CmiListReduce, CmiListReduceStruct, CmiGroupReduce, CmiGroupReduceStruct) are used for reductions over subsets of processors. They all require a CmiReductionID that the user must obtain in one of the ways described above. The user is also responsible that no two reductions use the same CmiReductionID simultaneously. The first two functions receive the subset description as processor list (pes) of size npes. The last two receive the subset description as a previously established CmiGroup (see 13.2.7).

13.2.9 Scheduling Messages

The scheduler queue is a powerful priority queue. The following functions can be used to place messages into the scheduler queue. These messages are treated very much like newly-arrived messages: when they reach the front of the queue, they trigger handler functions, just like messages transmitted with CMI functions. Note that unlike the CMI send functions, these cannot move messages across processors.

Every message inserted into the queue has a priority associated with it. Converse priorities are arbitrary-precision numbers between 0 and 1. Priorities closer to 0 get processed first, priorities closer to 1 get processed last. Arbitrary-precision priorities are very useful in AI search-tree applications. Suppose we have a heuristic suggesting that tree node N1 should be searched before tree node N2. We therefore designate that node N1 and its descendants will use high priorities, and that node N2 and its descendants will use lower priorities. We have effectively split the range of possible priorities in two. If several such heuristics fire in sequence, we can easily split the priority range in two enough times that no significant bits remain, and the search begins to fail for lack of meaningful priorities to assign. The solution is to use arbitrary-precision priorities, aka bitvector priorities.

These arbitrary-precision numbers are represented as bit-strings: for example, the bit-string “0011000101” represents the binary number (.0011000101). The format of the bit-string is as follows: the bit-string is represented as an array of unsigned integers. The most significant bit of the first integer contains the first bit of the bitvector. The remaining bits of the first integer contain the next 31 bits of the bitvector. Subsequent integers contain 32 bits each. If the size of the bitvector is not a multiple of 32, then the last integer contains 0 bits for padding in the least-significant bits of the integer.

Some people only want regular integers as priorities. For simplicity’s sake, we provide an easy way to convert integer priorities to Converse’s built-in representation.

In addition to priorities, you may choose to enqueue a message “LIFO” or “FIFO”. Enqueueing a message “FIFO” simply pushes it behind all the other messages of the same priority. Enqueueing a message “LIFO” pushes it in front of other messages of the same priority.

Messages sent using the CMI functions take precedence over everything in the scheduler queue, regardless of priority.

A recent addition to Converse scheduling mechanisms is the introduction of node-level scheduling designed to support low-overhead programming for the SMP clusters. These functions have “Node” in their names. All processors within the node has access to the node-level scheduler’s queue, and thus a message enqueued in a node-level queue may be handled by any processor within that node. When deciding about which message to process next, i.e. from processor’s own queue or from the node-level queue, a quick priority check is performed internally, thus a processor views scheduler’s queue as a single prioritized queue that includes messages directed at that processor and messages from the node-level queue sorted according to priorities.

```
void CsdEnqueueGeneral(void *Message, int strategy, int priobits, int *prioPtr)
```

This call enqueues a message to the processor’s scheduler’s queue, to be sorted according to its priority and the queuing strategy. The meaning of the priobits and prioPtr fields depend on the value of strategy, which are explained below.

```
void CsdNodeEnqueueGeneral(void *Message, int strategy, int priobits, int *prioPtr)
```

This call enqueues a message to the node-level scheduler’s queue, to be sorted according to its priority and the queuing strategy. The meaning of the priobits and prioPtr fields depend on the value of strategy, which can be any of the following:

- CQS_QUEUEING_BFIFO: the priobits and prioPtr point to a bit-string representing an arbitrary-precision priority. The message is pushed behind all other message of this priority.
- CQS_QUEUEING_BLIFO: the priobits and prioPtr point to a bit-string representing an arbitrary-precision priority. The message is pushed in front all other message of this priority.
- CQS_QUEUEING_IFIFO: the prioPtr is a pointer to a signed integer. The integer is converted to a bit-string priority, normalizing so that the integer zero is converted to the bit-string “1000...” (the “middle” priority). To be more specific, the conversion is performed by adding 0x80000000 to the integer, and then treating the resulting 32-bit quantity as a 32-bit bitvector priority. The message is pushed behind all other messages of this priority.
- CQS_QUEUEING_ILIFO: the prioPtr is a pointer to a signed integer. The integer is converted to a bit-string priority, normalizing so that the integer zero is converted to the bit-string “1000...” (the “middle” priority). To be more specific, the conversion is performed by adding 0x80000000 to the integer, and then treating the resulting 32-bit quantity as a 32-bit bitvector priority. The message is pushed in front of all other messages of this priority.
- CQS_QUEUEING_FIFO: the prioPtr and priobits are ignored. The message is enqueued with the middle priority “1000...”, and is pushed behind all other messages with this priority.
- CQS_QUEUEING_LIFO: the prioPtr and priobits are ignored. The message is enqueued with the middle priority “1000...”, and is pushed in front of all other messages with this priority.

Caution: the priority itself is *not copied* by the scheduler. Therefore, if you pass a pointer to a priority into the scheduler, you must not overwrite or free that priority until after the message has emerged from the scheduler’s queue. It is normal to actually store the priority *in the message itself*, though it is up to the user to actually arrange storage for the priority.

```
void CsdEnqueue(void *Message)
```

This macro is a shorthand for

```
CsdEnqueueGeneral(Message, CQS_QUEUEING_FIFO, 0, NULL)
```

provided here for backward compatibility.

```
void CsdNodeEnqueue(void *Message)
```

This macro is a shorthand for

```
CsdNodeEnqueueGeneral(Message, CQS_QUEUEING_FIFO, 0, NULL)
```

provided here for backward compatibility.

```
void CsdEnqueueFifo(void *Message)
```

This macro is a shorthand for

```
CsdEnqueueGeneral(Message, CQS_QUEUEING_FIFO, 0, NULL)
```

provided here for backward compatibility.

```
void CsdNodeEnqueueFifo(void *Message)
```

This macro is a shorthand for

```
CsdNodeEnqueueGeneral(Message, CQS_QUEUEING_FIFO, 0, NULL)
```

provided here for backward compatibility.

```
void CsdEnqueueLifo(void *Message)
```

This macro is a shorthand for

```
CsdEnqueueGeneral(Message, CQS_QUEUEING_LIFO, 0, NULL)
```

provided here for backward compatibility.

```
void CsdNodeEnqueueLifo(void *Message)
```

This macro is a shorthand for

```
CsdNodeEnqueueGeneral(Message, CQS_QUEUEING_LIFO, 0, NULL)
```

provided here for backward compatibility.

```
int CsdEmpty()
```

This function returns non-zero integer when the scheduler's processor-level queue is empty, zero otherwise.

```
int CsdNodeEmpty()
```

This function returns non-zero integer when the scheduler's node-level queue is empty, zero otherwise.

13.2.10 Polling for Messages

As we stated earlier, Converse messages trigger handler functions when they arrive. In fact, for this to work, the processor must occasionally poll for messages. When the user starts Converse, he can put it into one of several modes. In the normal mode, the message polling happens automatically. However *user-calls-scheduler* mode is designed to let the user poll manually. To do this, the user must use one of two polling functions: *CmiDeliverMsgs*, or *CsdScheduler*. *CsdScheduler* is more general, it will notice any Converse event. *CmiDeliverMsgs* is a lower-level function that ignores all events except for recently-arrived messages. (In particular, it ignores messages in the scheduler queue). You can save a tiny amount of overhead by using the lower-level function. We recommend the use of *CsdScheduler* for all applications except those that are using only the lowest level of Converse, the CMI. A third polling function,

`CmiDeliverSpecificMsg`, is used when you know the exact event you want to wait for: it does not allow any other event to occur.

In each iteration, a scheduler first looks for any message that has arrived from another processor, and delivers it. If there isn't any, it selects a message from the locally enqueued messages, and delivers it.

```
void CsdScheduleForever(void)
```

Extract and deliver messages until the scheduler is stopped. Raises the idle handling converse signals. This is the scheduler to use in most Converse programs.

```
int CsdScheduleCount(int n)
```

Extract and deliver messages until n messages have been delivered, then return 0. If the scheduler is stopped early, return n minus the number of messages delivered so far. Raises the idle handling converse signals.

```
void CsdSchedulePoll(void)
```

Extract and deliver messages until no more messages are available, then return. This is useful for running non-networking code when the networking code has nothing to do.

```
void CsdScheduler(int n)
```

If n is zero, call `CsdSchedulePoll`. If n is negative, call `CsdScheduleForever`. If n is positive, call `CsdScheduleCount(n)`.

```
int CmiDeliverMsgs(int MaxMsgs)
```

Retrieves messages from the network message queue and invokes corresponding handler functions for arrived messages. This function returns after either the network message queue becomes empty or after `MaxMsgs` messages have been retrieved and their handlers called. It returns the difference between total messages delivered and `MaxMsgs`. The handler is given a pointer to the message as its parameter.

```
void CmiDeliverSpecificMsg(int HandlerId)
```

Retrieves messages from the network queue and delivers the first message with its handler field equal to `HandlerId`. This functions leaves alone all other messages. It returns after the invoked handler function returns.

```
void CsdExitScheduler(void)
```

This call causes `CsdScheduler` to stop processing messages when control has returned back to it. The scheduler then returns to its calling routine.

13.2.11 The Timer

```
double CmiTimer(void)
```

Returns current value of the timer in seconds. This is typically the time spent since the `ConverseInit` call. The precision of this timer is the best available on the particular machine, and usually has at least microsecond accuracy.

13.2.12 Processor Ids

```
int CmiNumPe(void)
```

Returns the total number of processors on which the parallel program is being run.

```
int CmiMyPe(void)
```

Returns the logical processor identifier of processor on which the caller resides. A processor Id is between 0 and CmiNumPe() - 1.

Also see the calls in Section 13.2.13.

13.2.13 Global Variables and Utility functions

Different vendors are not consistent about how they treat global and static variables. Most vendors write C compilers in which global variables are shared among all the processors in the node. A few vendors write C compilers where each processor has its own copy of the global variables. In theory, it would also be possible to design the compiler so that each thread has its own copy of the global variables.

The lack of consistency across vendors, makes it very hard to write a portable program. The fact that most vendors make the globals shared is inconvenient as well, usually, you don't want your globals to be shared. For these reasons, we added "pseudoglobals" to Converse. These act much like C global and static variables, except that you have explicit control over the degree of sharing.

In this section we use the terms Node, PE, and User-level thread as they are used in Charm++, to refer to an OS process, a worker/communication thread, and a user-level thread, respectively. In the SMP mode of Charm++ all three of these are separate entities, whereas in non-SMP mode Node and PE have the same scope.

Converse PseudoGlobals

Three classes of pseudoglobal variables are supported: node-shared, processor-private, and thread-private variables.

Node-shared variables (Csv) are specific to a node. They are shared among all the PEs within the node.

PE-private variables (Cpv) are specific to a PE. They are shared by all the objects and Converse user-level threads on a PE.

Thread-private variables (Ctv) are specific to a Converse user-level thread. They are truly private.

There are five macros for each class. These macros are for declaration, static declaration, extern declaration, initialization, and access. The declaration, static and extern specifications have the same meaning as in C. In order to support portability, however, the global variables must be installed properly, by using the initialization macros. For example, if the underlying machine is a simulator for the machine model supported by Converse, then the thread-private variables must be turned into arrays of variables. Initialize and Access macros hide these details from the user. It is possible to use global variables without these macros, as supported by the underlying machine, but at the expense of portability.

Macros for node-shared variables:

```
CsvDeclare(type,variable)

CsvStaticDeclare(type,variable)

CsvExtern(type,variable)

CsvInitialize(type,variable)

CsvAccess(variable)
```

Macros for PE-private variables:

```

CpvDeclare(type,variable)

CpvStaticDeclare(type,variable)

CpvExtern(type,variable)

CpvInitialize(type,variable)

CpvAccess(variable)

```

Macros for thread-private variables:

```

CtvDeclare(type,variable)

CtvStaticDeclare(type,variable)

CtvExtern(type,variable)

CtvInitialize(type,variable)

CtvAccess(variable)

```

A sample code to illustrate the usage of the macros is provided in the example below. There are a few rules that the user must pay attention to: The `type` and `variable` fields of the macros must be a single word. Therefore, structures or pointer types can be used by defining new types with the `typedef`. In the sample code, for example, a `struct point` type is redefined with a `typedef` as `Point` in order to use it in the macros. Similarly, the access macros contain only the name of the global variable. Any indexing or member access must be outside of the macro as shown in the sample code (function `func1`). Finally, all the global variables must be installed before they are used. One way to do this systematically is to provide a module-init function for each file (in the sample code - `ModuleInit()`). The module-init functions of each file, then, can be called at the beginning of execution to complete the installations of all global variables.

File: Module1.c

```

typedef struct point
{
    float x,y;
} Point;

CpvDeclare(int, a);
CpvDeclare(Point, p);

void ModuleInit()
{
    CpvInitialize(int, a)
    CpvInitialize(Point, p);

    CpvAccess(a) = 0;
}

int func1()
{
    CpvAccess(p).x = 0;
    CpvAccess(p).y = CpvAccess(p).x + 1;
}

```

Utility Functions

To further simplify programming with global variables on shared memory machines, Converse provides the following functions and/or macros. (**Note:** These functions are defined on machines other than shared-memory machines also, and have the effect of only one processor per node and only one thread per processor.)

```
int CmiMyNode()
```

Returns the node number to which the calling processor belongs.

```
int CmiNumNodes()
```

Returns number of nodes in the system. Note that this is not the same as `CmiNumPes()`.

```
int CmiMyRank()
```

Returns the rank of the calling processor within a shared memory node.

```
int CmiNodeFirst(int node)
```

Returns the processor number of the lowest ranked processor on node `node`

```
int CmiNodeSize(int node)
```

Returns the number of processors that belong to the node `node`.

```
int CmiNodeOf(int pe)
```

Returns the node number to which processor `pe` belongs. Indeed, `CmiMyNode()` is a utility macro that is aliased to `CmiNodeOf(CmiMyPe())`.

```
int CmiRankOf(int pe)
```

Returns the rank of processor `pe` in the node to which it belongs.

Node-level Locks and other Synchronization Mechanisms

```
void CmiNodeBarrier()
```

Provide barrier synchronization at the node level, i.e. all the processors belonging to the node participate in this barrier.

```
typedef McDependentType CmiNodeLock
```

This is the type for all the node-level locks in Converse.

```
CmiNodeLock CmiCreateLock(void)
```

Creates, initializes and returns a new lock. Initially the lock is unlocked.

```
void CmiLock(CmiNodeLock lock)
```

Locks `lock`. If the `lock` has been locked by other processor, waits for `lock` to be unlocked.

```
void CmiUnlock(CmiNodeLock lock)
```

Unlocks `lock`. Processors waiting for the `lock` can then compete for acquiring `lock`.

```
int CmiTryLock(CmiNodeLock lock)
```

Tries to lock `lock`. If it succeeds in locking, it returns 0. If any other processor has already acquired the lock, it returns 1.

```
void CmiDestroyLock(CmiNodeLock lock)
```

Frees any memory associated with `lock`. It is an error to perform any operations with `lock` after a call to this function.

13.2.14 Input/Output

```
void CmiPrintf(char *format, arg1, arg2, ...)
```

This function does an atomic `printf()` on `stdout`. On machine with host, this is implemented on top of the messaging layer using asynchronous sends.

```
int CmiScanf(char *format, void *arg1, void *arg2, ...)
```

This function performs an atomic `scanf` from `stdin`. The processor, on which the caller resides, blocks for input. On machines with host, this is implemented on top of the messaging layer using asynchronous send and blocking receive.

```
void CmiError(char *format, arg1, arg2, ...)
```

This function does an atomic `printf()` on `stderr`. On machines with host, this is implemented on top of the messaging layer using asynchronous sends.

13.2.15 Spanning Tree Calls

Sometimes, it is convenient to view the processors/nodes of the machine as a tree. For this purpose, Converse defines a tree over processors/nodes. We provide functions to obtain the parent and children of each processor/node. On those machines where the communication topology is relevant, we arrange the tree to optimize communication performance. The root of the spanning tree (processor based or node-based) is always 0, thus the `CmiSpanTreeRoot` call has been eliminated.

```
int CmiSpanTreeParent(int procNum)
```

This function returns the processor number of the parent of `procNum` in the spanning tree.

```
int CmiNumSpanTreeChildren(int procNum)
```

Returns the number of children of `procNum` in the spanning tree.

```
void CmiSpanTreeChildren(int procNum, int *children)
```

This function fills the array `children` with processor numbers of children of `procNum` in the spanning tree.

```
int CmiNodeSpanTreeParent(int nodeNum)
```

This function returns the node number of the parent of `nodeNum` in the spanning tree.

```
int CmiNumNodeSpanTreeChildren(int nodeNum)
```

Returns the number of children of nodeNum in the spanning tree.

```
void CmiNodeSpanTreeChildren(int nodeNum, int *children)
```

This function fills the array children with node numbers of children of nodeNum in the spanning tree.

13.2.16 Isomalloc

It is occasionally useful to allocate memory at a globally unique virtual address. This is trivial on a shared memory machine (where every address is globally unique) but more difficult on a distributed memory machine. Isomalloc provides a uniform interface for allocating globally unique virtual addresses.

Isomalloc can thus be thought of as a software distributed shared memory implementation; except data movement between processors is explicit (by making a subroutine call), not on demand (by taking a page fault).

Isomalloc is useful when moving highly interlinked data structures from one processor to another, because internal pointers will still point to the correct locations, even on a new processor. This is especially useful when the format of the data structure is complex or unknown, as with thread stacks.

During Converse startup, a global reduction and broadcast takes place in order to find the intersection of available virtual address space across all logical nodes. If this operation causes an unwanted delay at startup or fails entirely for a system-specific reason, it can be disabled with the command line option `+no_isomalloc_sync`.

Effective management of the virtual address space across a distributed machine is a complex task that requires a certain level of organization. Therefore, Isomalloc is not well-suited to a fully dynamic API for allocating and migrating blocks on an individual basis. All allocations made using Isomalloc are managed as part of contexts corresponding to some unit of work, such as migratable threads. These contexts are migrated all at once. The total number of contexts must be determined before any use of Isomalloc takes place.

This API may evolve as new use cases emerge.

```
CmiIsomallocContext CmiIsomallocContextCreate(int myunit, int numunits)
```

Construct a context for a given unit of work, out of a total number of slots available. Successive calls to this function must always pass the same value for numunits. For example, if you are writing code using migratable threads, you must know at the beginning of execution what the maximum possible number of threads simultaneously in existence will be across the entire job, and each thread must have a globally unique ID. Multiple distinct sets of slots are currently unsupported.

This function in particular is likely to change in the future if new code using Isomalloc is developed, especially in the realm of interoperability between multiple simultaneous uses.

```
void CmiIsomallocContextDelete(CmiIsomallocContext ctx)
```

Destroy a given context, releasing all allocations owned by it and all virtual address space used by it.

```
void * CmiIsomallocContextMalloc(CmiIsomallocContext ctx, size_t size)
```

Allocate size bytes at a unique virtual address. Returns a pointer to the allocated region.

```
void * CmiIsomallocContextMallocAlign(CmiIsomallocContext ctx, size_t align, size_t_  
↪size)
```

Same as above, but with the alignment also specified. It must be a power of two.

```
void * CmiIsomallocContextCalloc(CmiIsomallocContext ctx, size_t nelem, size_t size)
```

Same as CmiIsomallocContextMalloc, but calloc instead of malloc.


```
void * CmiIsomallocContextRealloc(CmiIsomallocContext ctx, void * ptr, size_t size)
```

Same as `CmiIsomallocContextMalloc`, but `realloc` instead of `malloc`.

```
void CmiIsomallocContextFree(CmiIsomallocContext ctx, void * ptr)
```

Release the given block, which must have been previously allocated by the given `Isomalloc` context. It may also release the underlying virtual address range, which the system may subsequently reuse.

After a call to this function, any use of the freed space could corrupt the heap or cause a segmentation fault. It is illegal to free the same block more than once.

```
void CmiIsomallocContextPup(pup_er p, CmiIsomallocContext * ctxptr)
```

Pack/Unpack the given context. This routine can be used to move contexts across processors, save them to disk, or checkpoint them.

```
int CmiIsomallocContextGetLength(void * ptr)
```

Return the length, in bytes, of this `isomalloc`'d block.

```
int CmiIsomallocInRange(void * ptr)
```

Return 1 if the given address is in the virtual address space used by `Isomalloc`, 0 otherwise. `CmiIsomallocInRange(malloc(size))` is guaranteed to be zero; `CmiIsomallocInRange(CmiIsomallocContextMalloc(ctx, size))` is guaranteed to be one.

13.3 Threads

The calls in this chapter can be used to put together runtime systems for languages that support threads. This threads package, like most thread packages, provides basic functionality for creating threads, destroying threads, yielding, suspending, and awakening a suspended thread. In addition, it provides facilities whereby you can write your own thread schedulers.

13.3.1 Basic Thread Calls

```
typedef struct CthThreadStruct *CthThread;
```

This is an opaque type defined in `converse.h`. It represents a first-class thread object. No information is publicized about the contents of a `CthThreadStruct`.

```
typedef void (CthVoidFn) (void *);
```

This is a type defined in `converse.h`. It represents a function that returns nothing.

```
typedef CthThread (CthThFn) (void);
```

This is a type defined in `converse.h`. It represents a function that returns a `CthThread`.

```
CthThread CthSelf()
```

Returns the currently-executing thread. Note: even the initial flow of control that inherently existed when the program began executing `main` counts as a thread. You may retrieve that thread object using `CthSelf` and use it like any other.

```
CthThread CthCreate(CthVoidFn fn, void *arg, int size)
```

Creates a new thread object. The thread is not given control yet. To make the thread execute, you must push it into the scheduler queue, using `CthAwaken` below. When (and if) the thread eventually receives control, it will begin executing the specified function `fn` with the specified argument. The `size` parameter specifies the stack size in bytes, 0 means use the default size. Caution: almost all threads are created with `CthCreate`, but not all. In particular, the one initial thread of control that came into existence when your program was first `exec'd` was not created with `CthCreate`, but it can be retrieved (say, by calling `CthSelf` in `main`), and it can be used like any other `CthThread`.

```
CthThread CthCreateMigratable(CthVoidFn fn, void *arg, int size,   
↪CmiIsomallocContextctx)
```

Create a thread that can later be moved to other processors. Otherwise identical to `CthCreate`. An `Isomalloc` context is required for organized allocation management of the thread's stack, as well as any global variable privatization and heap interception in use.

This is only a hint to the runtime system; some threads implementations cannot migrate threads, others always create migratable threads. In these cases, `CthCreateMigratable` is equivalent to `CthCreate`.

```
CthThread CthPup(pup_er p, CthThread t)
```

Pack/Unpack a thread. This can be used to save a thread to disk, migrate a thread between processors, or checkpoint the state of a thread.

Only a suspended thread can be Pup'd. Only a thread created with `CthCreateMigratable` can be Pup'd.

```
void CthFree(CthThread t)
```

Frees thread `t`. You may ONLY free the currently-executing thread (yes, this sounds strange, it's historical). Naturally, the free will actually be postponed until the thread suspends. To terminate itself, a thread calls `CthFree(CthSelf())`, then gives up control to another thread.

```
void CthSuspend()
```

Causes the current thread to stop executing. The suspended thread will not start executing again until somebody pushes it into the scheduler queue again, using `CthAwaken` below. Control transfers to the next task in the scheduler queue.

```
void CthAwaken(CthThread t)
```

Pushes a thread into the scheduler queue. Caution: a thread must only be in the queue once. Pushing it in twice is a crashable error.

```
void CthAwakenPrio(CthThread t, int strategy, int priobits, int *prio)
```

Pushes a thread into the scheduler queue with priority specified by `priobits` and `prio` and queueing strategy `strategy`. Caution: a thread must only be in the queue once. Pushing it in twice is a crashable error. `prio` is not copied internally, and is used when the scheduler dequeues the message, so it should not be reused until then.

```
void CthYield()
```

This function is part of the scheduler-interface. It simply executes `{ CthAwaken(CthSelf()); CthSuspend(); }`. This combination gives up control temporarily, but ensures that control will eventually return.

```
void CthYieldPrio(int strategy, int priobits, int *prio)
```

This function is part of the scheduler-interface. It simply executes `{CthAwakenPrio(CthSelf(), strategy, priobits, prio); CthSuspend();}` This combination gives up control temporarily, but ensures that control will eventually return.

```
CthThread CthGetNext(CthThread t)
```

Each thread contains space for the user to store a “next” field (the functions listed here pay no attention to the contents of this field). This field is typically used by the implementors of mutexes, condition variables, and other synchronization abstractions to link threads together into queues. This function returns the contents of the next field.

```
void CthSetNext(CthThread t, CthThread next)
```

Each thread contains space for the user to store a “next” field (the functions listed here pay no attention to the contents of this field). This field is typically used by the implementors of mutexes, condition variables, and other synchronization abstractions to link threads together into queues. This function sets the contents of the next field.

13.3.2 Thread Scheduling and Blocking Restrictions

Converse threads use a scheduler queue, like any other threads package. We chose to use the same queue as the one used for Converse messages (see Section 13.2.9). Because of this, thread context-switching will not work unless there is a thread polling for messages. A rule of thumb, with Converse, it is best to have a thread polling for messages at all times. In Converse’s normal mode (see Section 13.1), this happens automatically. However, in user-calls-scheduler mode, you must be aware of it.

There is a second caution associated with this design. There is a thread polling for messages (even in normal mode, it’s just hidden in normal mode). The continuation of your computation depends on that thread — you must not block it. In particular, you must not call blocking operations in these places:

- In the code of a Converse handler (see Sections 13.2.2 and 13.2.3).
- In the code of the Converse start-function (see section 13.1).

These restrictions are usually easy to avoid. For example, if you wanted to use a blocking operation inside a Converse handler, you would restructure the code so that the handler just creates a new thread and returns. The newly-created thread would then do the work that the handler originally did.

13.3.3 Thread Scheduling Hooks

Normally, when you CthAwaken a thread, it goes into the primary ready-queue: namely, the main Converse queue described in Section 13.2.9. However, it is possible to hook a thread to make it go into a different ready-queue. That queue doesn’t have to be priority-queue: it could be FIFO, or LIFO, or in fact it could handle its threads in any complicated order you desire. This is a powerful way to implement your own scheduling policies for threads.

To achieve this, you must first implement a new kind of ready-queue. You must implement a function that inserts threads into this queue. The function must have this prototype:

```
void awakenfn(CthThread t, int strategy, int priobits, int *prio);
```

When a thread suspends, it must choose a new thread to transfer control to. You must implement a function that makes the decision: which thread should the current thread transfer to. This function must have this prototype:

```
CthThread choosefn();
```

Typically, the choosefn would choose a thread from your ready-queue. Alternately, it might choose to always transfer control to a central scheduling thread.

You then configure individual threads to actually use this new ready-queue. This is done using CthSetStrategy:

```
void CthSetStrategy(CthThread t, CthAwkFn awakenfn, CthThFn choosefn)
```

Causes the thread to use the specified `awakenfn` whenever you `CthAwaken` it, and the specified `choosefn` whenever you `CthSuspend` it.

`CthSetStrategy` alters the behavior of `CthSuspend` and `CthAwaken`. Normally, when a thread is awakened with `CthAwaken`, it gets inserted into the main ready-queue. Setting the thread's `awakenfn` will cause the thread to be inserted into your ready-queue instead. Similarly, when a thread suspends using `CthSuspend`, it normally transfers control to some thread in the main ready-queue. Setting the thread's `choosefn` will cause it to transfer control to a thread chosen by your `choosefn` instead.

You may reset a thread to its normal behavior using `CthSetStrategyDefault`:

```
void CthSetStrategyDefault(CthThread t)
```

Restores the value of `awakenfn` and `choosefn` to their default values. This implies that the next time you `CthAwaken` the specified thread, it will be inserted into the normal ready-queue.

Keep in mind that this only resolves the issue of how threads get into your ready-queue, and how those threads suspend. To actually make everything “work out” requires additional planning: you have to make sure that control gets transferred to everywhere it needs to go.

Scheduling threads may need to use this function as well:

```
void CthResume(CthThread t)
```

Immediately transfers control to thread `t`. This routine is primarily intended for people who are implementing schedulers, not for end-users. End-users should probably call `CthSuspend` or `CthAwaken` (see below). Likewise, programmers implementing locks, barriers, and other synchronization devices should also probably rely on `CthSuspend` and `CthAwaken`.

A final caution about the `choosefn`: it may only return a thread that wants the CPU, eg, a thread that has been awakened using the `awakenfn`. If no such thread exists, if the `choosefn` cannot return an awakened thread, then it must not return at all: instead, it must wait until, by means of some pending IO event, a thread becomes awakened (pending events could be asynchronous disk reads, networked message receptions, signal handlers, etc). For this reason, many schedulers perform the task of polling the IO devices as a side effect. If handling the IO event causes a thread to be awakened, then the `choosefn` may return that thread. If no pending events exist, then all threads will remain permanently blocked, the program is therefore done, and the `choosefn` should call `exit`.

There is one minor exception to the rule stated above (“the scheduler may not resume a thread unless it has been declared that the thread wants the CPU using the `awakenfn`”). If a thread `t` is part of the scheduling module, it is permitted for the scheduling module to resume `t` whenever it so desires: presumably, the scheduling module knows when its threads want the CPU.

13.4 Timers, Periodic Checks, and Conditions

This module provides functions that allow users to insert hooks, i.e. user-supplied functions, that are called by the system at as specific conditions arise. These conditions differ from UNIX signals in that they are raised synchronously, via a regular function call; and that a single condition can call several different functions.

The system-defined conditions are:

CcdPROCESSOR_BEGIN_IDLE Raised when the scheduler first finds it has no messages to execute. That is, this condition is raised at the trailing edge of the processor utilization graph.

CcdPROCESSOR_STILL_IDLE Raised when the scheduler subsequently finds it still has no messages to execute. That is, this condition is raised while the processor utilization graph is flat.

CcdPROCESSOR_LONG_IDLE This is an extension of **CcdPROCESSOR_STILL_IDLE** for a relatively longer period of time. It is raised when the scheduler finds that it doesn't have any messages to execute for a long period of time. The default **LONG_IDLE** time is 10 seconds. However, it is customizable using a user passed runtime flag `+longIdleThresh <long idle time in seconds>`. This feature is useful for debugging hangs in applications. It can allow the user to add a user defined function as a hook to execute when the program goes into a long idle state, typically seen during hangs. The test program `tests/charm++/longIdle` illustrates the usage of **CcdPROCESSOR_LONG_IDLE**. Since the usage of **CcdPROCESSOR_LONG_IDLE** uses additional timers in the scheduler loop, it is only enabled with error checking builds and requires the user to build the target with `--enable-error-checking`.

CcdPROCESSOR_BEGIN_BUSY Raised when a message first arrives on an idle processor. That is, raised on the rising edge of the processor utilization graph.

CcdPERIODIC The scheduler attempts to raise this condition every few milliseconds. The scheduling for this and the other periodic conditions is nonpreemptive, and hence may be delayed until the current entry point is finished.

CcdPERIODIC_10ms Raised every 10ms (at 100Hz).

CcdPERIODIC_100ms Raised every 100ms (at 10Hz).

CcdPERIODIC_1second Raised once per second.

CcdPERIODIC_10second Raised once every 10 seconds.

CcdPERIODIC_1minute Raised once per minute.

CcdPERIODIC_10minute Raised once every 10 minutes.

CcdPERIODIC_1hour Raised once every hour.

CcdPERIODIC_12hour Raised once every twelve hours.

CcdPERIODIC_1day Raised once every day.

CcdSCHEDLOOP Raised at every scheduler loop. Use with caution to avoid adding significant overhead to the scheduler.

CcdQUIESCENCE Raised when the quiescence detection system has determined that the system is quiescent.

CcdSIGUSR1 Raised when the system receives the UNIX signal **SIGUSR1**. Be aware that this condition is thus raised asynchronously, from within a signal handler, and all the usual signal handler restrictions apply.

CcdSIGUSR2 Raised when the system receives the UNIX signal **SIGUSR2**.

CcdUSER The system never raises this or any larger conditions. They can be used by the user for application-specific use.

CcdUSERMAX All conditions from **CcdUSER** to **CcdUSERMAX** (inclusive) are available.

```
int CcdCallOnCondition(int condnum, CcdVoidFn fnp, void* arg)
```

This call instructs the system to call the function indicated by the function pointer `fnp`, with the specified argument `arg`, when the condition indicated by `condnum` is raised next. Multiple functions may be registered for the same condition number. `CcdVoidFn` is a function pointer with the signature `void fnp(void *userParam, double curWallTime)`

```
int CcdCallOnConditionKeep(int condnum, CcdVoidFn fnp, void* arg)
```

As above, but the association is permanent- the given function will be called again whenever this condition is raised. Returns an index that may be used to cancel the association later.

```
void CcdCancelCallOnCondition(int condnum, int idx)
```

```
void CcdCancelCallOnConditionKeep(int condnum, int idx)
```

Delete the given index from the list of callbacks for the given condition. The corresponding function will no longer be called when the condition is raised. Note that it is illegal to call these two functions to cancel callbacks from within ccd callbacks.

```
double CcdRaiseCondition(int condNum)
```

When this function is called, it invokes all the functions whose pointers were registered for the `condNum` via a *prior* call to `CcdCallOnCondition` or `CcdCallOnConditionKeep`. The function internally calls `CmiWallTimer` and returns this value. When using `CcdRaiseCondition`, the return value can be used to determine the current walltime avoiding an additional call to `CmiWallTimer`. However, it is important to note that the walltime value returned by `CcdRaiseCondition` could be stale by the time it is returned since registered functions are executed between the timer call and the return. For this reason, this walltime value returned should be used in situations where an exact or current timer value is not desired.

```
void CcdCallFnAfter(CcdVoidFn fnp, void* arg, double msLater)
```

This call registers a function via a pointer to it, `fnp`, that will be called at least `msLater` milliseconds later. The registered function `fnp` is actually called the first time the scheduler gets control after `deltaT` milliseconds have elapsed. The default polling resolution for timed callbacks is 5 ms.

```
double CcdSetResolution(double newResolution)
```

This call sets the polling resolution for completion of timed callbacks. `newResolution` is the updated time in seconds. The default polling resolution for timed callbacks is 5 ms. The resolution cannot be any longer than this but it can be set arbitrarily short. Shorter resolution times can result in a performance decrease due to more time being spent polling for callbacks but may be preferred in cases where these need to be triggered quickly and/or are on the critical path of an application. This function also returns the old resolution in seconds in case it needs to be reset to a non-default value.

```
double CcdResetResolution()
```

This call returns the time based callback polling resolution to its default, 5 milliseconds. It returns the previously set resolution in seconds.

```
double CcdIncreaseResolution(double newResolution)
```

This is a “safe” version of `CcdSetResolution` that only ever sets the resolution to a shorter time. The same caveats about short polling times affecting performance still apply. This function returns the previous (and potentially current, if it was shorter than `newResolution`.) resolution in seconds.

13.5 Converse Client-Server Interface

This note describes the Converse client-server (CCS) module. This module enables Converse programs to act as parallel servers, responding to requests from (non-Converse) programs across the internet.

The CCS module is split into two parts- client and server. The server side is the interface used by a Converse program; the client side is used by arbitrary (non-Converse) programs. The following sections describe both these parts.

A CCS client accesses a running Converse program by talking to a `server-host`, which receives the CCS requests and relays them to the appropriate processor. The `server-host` is `charmrun` for `netlrts-` versions, and is the first processor for all other versions.

13.5.1 CCS: Starting a Server

A Converse program is started using

```
$ charmrun pgmname +pN charmrun-opts pgm-opts
```

charmrun also accepts the CCS options:

`++server:` open a CCS server on any TCP port number

`++server-port=port:` open the given TCP port as a CCS server

`++server-auth=authfile:` accept authenticated queries

As the parallel job starts, it will print a line giving the IP address and TCP port number of the new CCS server. The format is: “ccs: Server IP = *ip*, Server port = *port* \$”, where *ip* is a dotted decimal version of the server IP address, and *port* is the decimal port number.

13.5.2 CCS: Client-Side

A CCS client connects to a CCS server, asks a server PE to execute a pre-registered handler, and receives the response data. The CCS client may be written in any language (see CCS network protocol, below), but a C interface (files “ccs-client.c” and “ccs-client.h”) and Java interface (file “CcsServer.java”) are available in the charm include directory.

The C routines use the `skt_abort` error-reporting strategy; see “sockRoutines.h” for details. The C client API is:

```
void CcsConnect(CcsServer *svr, char *host, int port);
```

Connect to the

given CCS server. `svr` points to a pre-allocated `CcsServer` structure.

```
void CcsConnectIp(CcsServer *svr, int ip, int port);
```

As above, but a numeric IP is specified.

```
int CcsNumNodes(CcsServer *svr);
int CcsNumPes(CcsServer *svr);
int CcsNodeFirst(CcsServer *svr, int node);
int CcsNodeSize(CcsServer *svr, int node);
```

These functions return information about the parallel machine; they are equivalent to the Converse calls `CmiNumNodes`, `CmiNumPes`, `CmiNodeFirst`, and `CmiNodeSize`.

```
void CcsSendRequest(CcsServer *svr, char *hdlrID, int pe, unsigned int
size, const char *msg);
```

Ask the server to execute the handler `hdlrID` on the given processor. The handler is passed the given data as a message. The data may be in any desired format (including binary).

```
int CcsSendBroadcastRequest(CcsServer *svr, const char *hdlrID, int
size, const void *msg);
```

As `CcsSendRequest`, only that the handler `hdlrID` is invoked on all processors.

```
int CcsSendMulticastRequest(CcsServer *svr, const char *hdlrID,
int npes, int *pes, int size, const void *msg);
```

As CcsSendRequest, only that the handler hdlrID is invoked on the processors specified in the array pes (of size npes).

```
int CcsRecvResponse(CcsServer *svr, unsigned int maxsize,
char *recvBuffer, int timeout);
```

Receive a response to the previous request in-place. Timeout gives the number of seconds to wait before returning 0; otherwise the number of bytes received is returned.

```
int CcsRecvResponseMsg(CcsServer *svr, unsigned int *retSize,
char **newBuf, int timeout);
```

As above, but receive a variable-length response. The returned buffer must be free()'d after use.

```
int CcsProbe(CcsServer *svr);
```

Return 1 if a response is available; otherwise 0.

```
void CcsFinalize(CcsServer *svr);
```

Closes connection and releases server.

The Java routines throw an IOException on network errors. Use javadoc on CcsServer.java for the interface, which mirrors the C version above.

13.5.3 CCS: Server-Side

Once a request arrives on a CCS server socket, the CCS server runtime looks up the appropriate handler and calls it. If no handler is found, the runtime prints a diagnostic and ignores the message.

CCS calls its handlers in the usual Converse fashion- the request data is passed as a newly-allocated message, and the actual user data begins CmiMsgHeaderSizeBytes into the message. The handler is responsible for CmiFree'ing the passed-in message.

The interface for the server side of CCS is included in "converse.h"; if CCS is disabled (in conv-mach.h), all CCS routines become macros returning 0.

The handler registration interface is:

```
void CcsUseHandler(char *id, int hdlr);

int CcsRegisterHandler(char *id, CmiHandler fn);
```

Associate this handler ID string with this function. hdlr is a Converse handler index; fn is a function pointer. The ID string cannot be more than 32 characters, including the terminating NULL.

After a handler has been registered to CCS, the user can also setup a merging function. This function will be passed in to CmiReduce to combine replies to multicast and broadcast requests.

```
void CcsSetMergeFn(const char *name, CmiReduceMergeFn newMerge);
```

Associate the given merge function to the CCS identified by id. This will be used for CCS request received as broadcast or multicast.

These calls can be used from within a CCS handler:

```
int CcsEnabled(void);
```

Return 1 if CCS routines are available (from conv-mach.h). This routine does not determine if a CCS server port is actually open.


```
int CcsIsRemoteRequest(void);
```

Return 1 if this handler was called via CCS; 0 if it was called as the result of a normal Converse message.

```
void CcsCallerId(skt_ip_t *pip, unsigned int *pport);
```

Return the IP address and TCP port number of the CCS client that invoked this method. Can only be called from a CCS handler invoked remotely.

```
void CcsSendReply(int size, const void *reply);
```

Send the given data back to the client as a reply. Can only be called from a CCS handler invoked remotely. In case of broadcast or multicast CCS requests, the handlers in all processors involved must call this function.

```
CcsDelayedReply CcsDelayReply(void);
```

Allows a CCS reply to be delayed until after the handler has completed. Returns a token used below.

```
void CcsSendDelayedReply(CcsDelayedReply d, int size, const void *reply);
```

Send a CCS reply for the given request. Unlike CcsSendReply, can be invoked from any handler on any processor.

13.5.4 CCS: system handlers

The CCS runtime system provides several built-in CCS handlers, which are available in any Converse job:

ccs_getinfo Takes an empty message, responds with information about the parallel job. The response is in the form of network byte order (big-endian) 4-byte integers: first the number of parallel nodes, then the number of processors on each node. This handler is invoked by the client routine CcsConnect.

ccs_killport Allows a client to be notified when a parallel run exits (for any reason). Takes one network byte order (big-endian) 4-byte integer: a TCP port number. The runtime writes “die n” to this port before exiting. There is no response data.

perf_monitor Takes an empty message, responds (after a delay) with performance data. When CMK_WEB_MODE is enabled in conv-mach.h, the runtime system collects performance data. Every 2 seconds, this data is collected on processor 0 and sent to any clients that have invoked perf_monitor on processor 0. The data is returned in ASCII format with the leading string “perf”, and for each processor the current load (in percent) and scheduler message queue length (in messages). Thus a heavily loaded, two-processor system might reply with the string “perf 98 148230 100 385401”.

13.5.5 CCS: network protocol

This information is provided for completeness and clients written in non-C, non-Java languages. The client and server APIs above are the preferred way to use CCS.

A CCS request arrives as a new TCP connection to the CCS server port. The client speaks first, sending a request header and then the request data. The server then sends the response header and response data, and closes the connection. Numbers are sent as network byte order (big-endian) 4-byte integers- network binary integers.

The request header has three fields: the number of bytes of request data, the (0-based) destination processor number, and the CCS handler identifier string. The byte count and processor are network binary integers (4 bytes each), the CCS handler ID is zero-terminated ASCII text (32 bytes); for a total request header length of 40 bytes. The remaining request data is passed directly to the CCS handler.

The response header consists of a single network binary integer- the length in bytes of the response data to follow. The header is thus 4 bytes long. If there is no response data, this field has value 0.

13.5.6 CCS: Authentication

By default, CCS provides no authentication- this means any client anywhere on the internet can interact with the server. *authfile*, passed to '++server-auth', is a configuration file that enables authentication and describes the authentication to perform.

The configuration file is line-oriented ASCII text, consisting of security level / key pairs. The security level is an integer from 0 (the default) to 255. Any security levels not listed in the file are disallowed.

The key is the 128-bit secret key used to authenticate CCS clients for that security level. It is either up to 32 hexadecimal digits of key data or the string "OTP". "OTP" stands for One Time Pad, which will generate a random key when the server is started. This key is printed out at job startup with the format "CCS_OTP_KEY:math:> Level *i* key: *hexdigits*" where *i* is the security level in decimal and *hexdigits* is 32 hexadecimal digits of key data.

For example, a valid CCS authentication file might consist of the single line "0 OTP", indicating that the default security level 0 requires a randomly generated key. All other security levels are disallowed.

13.6 Converse One Sided Communication Interface

This chapter deals with one sided communication support in converse. It is imperative to provide a one-sided communication interface to take advantage of the hardware RDMA facilities provided by a lot of NIC cards. Drivers for these hardware provide or promise to soon provide capabilities to use this feature.

Converse provides an implementation which wraps the functionality provided by different hardware and presents them as a uniform interface to the programmer. For machines which do not have a one-sided hardware at their disposal, these operations are emulated through converse messages.

Converse provides the following types of operations to support one-sided communication.

13.6.1 Registering / Unregistering Memory for RDMA

The interface provides functions to register(pin) and unregister(unpin) memory on the NIC hardware. The emulated version of these operations do not do anything.

```
int CmiRegisterMemory(void *addr, unsigned int size);
```

This function takes an allocated memory at starting address *addr* of length *size* and registers it with the hardware NIC, thus making this memory DMAable. This is also called pinning memory on the NIC hardware, making remote DMA operations on this memory possible. This directly calls the hardware driver function for registering the memory region and is usually an expensive operation, so should be used sparingly.

```
int CmiUnRegisterMemory(void *addr, unsigned int size);
```

This function unregisters the memory at starting address *addr* of length *size*, making it no longer DMAable. This operation corresponds to unpinning memory from the NIC hardware. This is also an expensive operation and should be sparingly used.

For certain machine layers which support a DMA, we support the function `void *CmiDMAAlloc(int size);`

This operation allocates a memory region of length *size* from the DMAable region on the NIC hardware. The memory region returned is pinned to the NIC hardware. This is an alternative to `CmiRegisterMemory` and is implemented only for hardwares that support this.

13.6.2 RDMA operations (Get / Put)

This section presents functions that provide the actual RDMA operations. For hardware architectures that support these operations these functions provide a standard interface to the operations, while for NIC architectures that do not support RDMA operations, we provide an emulated implementation. There are three types of NIC architectures based on how much support they provide for RDMA operations:

- Hardware support for both *Get* and *Put* operations.
- Hardware support for one of the two operations, mostly for *Put*. For these the other RDMA operation is emulated by using the operation that is implemented in hardware and extra messages.
- No hardware support for any RDMA operation. For these, both the RDMA operations are emulated through messages.

There are two different sets of RDMA operations

- The first set of RDMA operations return an opaque handle to the programmer, which can only be used to verify if the operation is complete. This suits AMPI better and closely follows the idea of separating communication from synchronization. So, the user program needs to keep track of synchronization.
- The second set of RDMA operations do not return anything, instead they provide a callback when the operation completes. This suits nicely the charm++ framework of sending asynchronous messages. The handler(callback) will be automatically invoked when the operation completes.

For machine layer developer: Internally, every machine layer is free to create a suitable data structure for this purpose. This is the reason this has been kept opaque from the programmer.

```
void *CmiPut(unsigned int sourceId, unsigned int targetId, void
*Saddr, void *Taadr, unsigned int size);
```

This function is pretty self explanatory. It puts the memory location at Saddr on the machine specified by sourceId to Taddr on the machine specified by targetId. The memory region being RDMA'ed is of length size bytes.

```
void *CmiGet(unsigned int sourceId, unsigned int targetId, void
*Saddr, void *Taadr, unsigned int size);
```

Similar to CmiPut except the direction of the data transfer is opposite; from target to source.

```
void CmiPutCb(unsigned int sourceId, unsigned int targetId, void
*Saddr, void *Taddr, unsigned int size, CmiRdmaCallbackFn fn, void
*param);
```

Similar to CmiPut except a callback is called when the operation completes.

```
void CmiGetCb(unsigned int sourceId, unsigned int targetId, void
*Saddr, void *Taddr, unsigned int size, CmiRdmaCallbackFn fn, void
*param);
```

Similar to CmiGet except a callback is called when the operation completes.

13.6.3 Completion of RDMA operation

This section presents functions that are used to check for completion of an RDMA operation. The one sided communication operations are asynchronous, thus there needs to be a mechanism to verify for completion. One mechanism is for the programmer to check for completion. The other mechanism is through callback functions registered during the RDMA operations.

```
int CmiWaitTest(void *obj);
```

This function takes this RDMA handle and verifies if the operation corresponding to this handle has completed.

A typical usage of this function would be in AMPI when there is a call to `AMPIWait`. The implementation should call the `CmiWaitTest` for all pending RDMA operations in that window.

13.7 Random Number Generation

Converse includes support for random number generation using a 64-bit Linear Congruential Generator (LCG). The user can choose between using a supplied default stream shared amongst all chares on the processor, or creating a private stream. Note that there is a limit on the number of private streams, which at the time of writing was 15,613.

```
struct CrnStream;
```

This structure contains the current state of a random number stream. The user is responsible for allocating the memory for this structure.

13.7.1 Default Stream Calls

```
void CrnSrand(int seed);
```

Seeds the default random number generator with `seed`.

```
int CrnRand(void);
```

Returns the next random number in the default stream as an integer.

```
int CrnDrand(void);
```

Returns the next random number in the default stream as a double.

13.7.2 Private Stream Calls

```
void CrnInitStream(CrnStream *dest, int seed, int type);
```

Initializes a new stream with its initial state stored in `dest`. The user must supply a seed in `seed`, as well as the type of the stream, where the type can be 0, 1, or 2.

```
double CrnDouble(CrnStream *genptr);
```

Returns the next random number in the stream whose state is given by `genptr`; the number is returned as a double.

```
double CrnInt(CrnStream *genptr);
```

Returns the next random number in the stream whose state is given by `genptr`; the number is returned as an integer.

```
double CrnFloat(CrnStream *genptr);
```

Returns the next random number in the stream whose state is given by `genptr`; the number is returned as a float. (Note: This function is exactly equivalent to `(float) CrnDouble(genptr);`)

13.8 Converse Persistent Communication Interface

This chapter deals with persistent communication support in converse. It is used when point-to-point message communication is called repeatedly to avoid redundancy in setting up the message each time it is sent. In the message-driven model like charm, the sender will first notify the receiver that it will send message to it, the receiver will create handler to record the message size and malloc the address for the upcoming message and send that information back to the sender, then if the machine have one-sided hardware, it can directly put the message into the address on the receiver.

Converse provides an implementation which wraps the functionality provided by different hardware and presents them as a uniform interface to the programmer. For machines which do not have a one-sided hardware at their disposal, these operations are emulated through converse messages.

Converse provides the following types of operations to support persistent communication.

13.8.1 Create / Destroy Persistent Handler

The interface provides functions to create and destroy handler on the processor for use of persistent communication.

```
PersistentHandle CmiCreatePersistent(int destPE, int maxBytes);
```

This function creates a persistent communication handler with dest PE and maximum bytes for this persistent communication. Machine layer will send message to destPE and setup a persistent communication. A buffer of size maxBytes is allocated in the destination PE.

```
PersistentReq CmiCreateReceiverPersistent(int maxBytes);
PersistentHandle CmiRegisterReceivePersistent(PersistentReq req);
```

Alternatively, a receiver can initiate the setting up of persistent communication. At receiver side, user calls CmiCreateReceiverPersistent() which returns a temporary handle type - PersistentRecvHandle. Send this handle to the sender side and the sender should call CmiRegisterReceivePersistent() to setup the persistent communication. The function returns a PersistentHandle which can then be used for the persistent communication.

```
void CmiDestroyPersistent(PersistentHandle h);
```

This function destroys a persistent communication specified by PersistentHandle h.

```
void CmiDestroyAllPersistent();
```

This function will destroy all persistent communication on the local processor.

13.8.2 Persistent Operation

This section presents functions that uses persistent handler for communications.

```
void CmiUsePersistentHandle(PersistentHandle *p, int n)
```

This function will ask Charm machine layer to use an array of PersistentHandle "p"(array size of n) for all the following communication. Calling with p=NULL will cancel the persistent communication. n=1 is for sending message to each Chare, n>1 is for message in multicast-one PersistentHandle for each PE.

Converse Extensions Library

Contents

- *Converse Extensions Library*
 - *Introduction*
 - *Tag Matching*
 - *Converse Master-Slave Library*
 - * *Introduction*
 - * *Available Functions*
 - * *Example Program*
 - *Data Structures*
 - * *Queues, Lists, FIFOs etc.*
 - *Converse Pseudorandom Number Generator*
 - *Automatic Parameter Marshalling*
 - * *CPM Basics*
 - * *CPM Packing and Unpacking*
 - * *Inventing New Types of CpmDestinations*
 - *Load Balancing*
 - * *Using Converse Load Balancers*
 - * *How to Write a Load Balancer for Converse/Charm++*
 - * *A Sample Load Balancer*
 - *Futures*

- *Converse-POSIX threads*
 - * *Pthreads and Converse*
 - * *Suppressing Name Conflicts*
 - * *Interoperating with Other Thread Packages*
 - * *Preemptive Context Switching*
 - * *Limits on Blocking Operations in main*
 - * *CpthreadModuleInit*
- *Parallel Arrays of Threads*
 - * *Creating Arrays of Threads*
 - * *Mapping Functions for Arrays of Threads*
 - * *Thread Functions for Arrays of Threads*
 - * *Sending Messages to Threads*
 - * *Performing Reductions over Array Elements*

14.1 Introduction

The Converse Extensions Library is a collection of modules that have been implemented on top of the Converse API. Each of these modules was deemed potentially useful to other Converse users, thus, we distribute these modules along with Converse as a convenience. *You don't need to read any part of this manual to use Converse.*

14.2 Tag Matching

The message manager is a data structure that can be used to put together runtime systems for languages that support tag-based message retrieval.

The purpose of the message manager is to store, index, and retrieve messages according to a set of integer tags. It provides functions to create tables, functions to insert messages into tables (specifying their tags), and functions to selectively retrieve messages from tables according to their tags. Wildcard tags can be specified in both storage and retrieval.

To use the message manager, you must include `converse.h` and link with the Converse library.

In actuality, the term “message manager” is unnecessarily specific. The message manager can store and retrieve arbitrary pointers according to a set of tags. The pointers do *not* necessarily need to be pointers to Converse messages. They can be pointers to anything.

```
typedef struct CmmTableStruct *CmmTable
```

This opaque type is defined in `converse.h`. It represents a table which can be used to store messages. No information is publicized about the format of a `CmmTableStruct`.

```
#define CmmWildCard (-1)
```

This `#define` is in `converse.h`. The tag `-1` is the “wild card” for the tag-based lookup functions in the message manager.

```
CmmTable CmmNew();
```


This function creates a new message-table and returns it.

```
void CmmPut(CmmTable t, int ntags, int *tags, void *msg)
```

This function inserts a message into a message table, along with an array of tags. `ntags` specifies the length of the `tags` array. The `tags` array contains the tags themselves. `msg` and `t` specify the message and table, respectively.

```
void *CmmGet(CmmTable t, int ntags, int *tags, int *ret_tags)
```

This function looks up a message from a message table. A message will be retrieved that “matches” the specified `tags` array. If a message is found that “matches”, the tags with which it was stored are copied into the `ret_tags` array, a pointer to the message will be returned, and the message will be deleted from the table. If no match is found, 0 will be returned.

To “match”, the array `tags` must be of the same length as the stored array. Similarly, all the individual tags in the stored array must “match” the tags in the `tags` array. Two tags match if they are equal to each other, or if either tag is equal to `CmmWildcard` (this means one can store messages with wildcard tags, making it easier to find those messages on retrieval).

```
void *CmmProbe(CmmTable t, int ntags, int *tags, int *ret_tags)
```

This function is identical to `CmmGet` above, except that the message is not deleted from the table.

```
void CmmFree(CmmTable t);
```

This function frees a message-table `t`. **WARNING:** It also frees all the messages that have been inserted into the message table. It assumes that the correct way to do this is to call `CmiFree` on the message. If this assumption is incorrect, a crash will occur. The way to avoid this problem is to remove and properly dispose all the messages in a table before disposing the table itself.

14.3 Converse Master-Slave Library

14.3.1 Introduction

CMS is the implementation of the master-slave (or manager-worker or agenda) parallel programming paradigm on top of Converse.

14.3.2 Available Functions

Following functions are available in this library:

```
typedef int (*CmsWorkerFn) (void *, void *);
```

Prototype for the worker function. See below.

```
typedef int (*CmsConsumerFn) (void *, int);
```

Prototype for the consumer function. See below.

```
void CmsInit(CmsWorkerFn worker, int max);
```

This function must be called before firing any tasks for the workers. `max` is the largest possible number of tasks you will fire before calling either `CmsAwaitResponses` or `CmsProcessResponses` next. (So the system know how many it may have to buffer).

```
int worker(void *t, void **r)
```

The user writes this function. Its name does not have to be `worker`; It can be anything. `worker` can be any function that the use writes to perform the task on the slave processors. It must allocate and compute the response data structure, and return a pointer to it, by assigning to `r`; It must also return the size of the response data structure as its return value.

```
void CmsFireTask(int ref, void * t, int size)
```

Creates task to be worked on by a worker. The task description is pointed to by t, and goes on for size bytes. ref must be a unique serial number between 0 and max (see CmsInit).

```
void CmsAwaitResponses(void);
```

This call allows the system to use processor 0 as a worker. It returns after all the tasks have sent back their responses. The responses themselves can be extracted using CmsGetResponse.

```
void *CmsGetResponse(int ref);
```

Extracts the response associated with the reference number ref from the system's buffers.

```
void CmsProcessResponses(CmsConsumerFn consumer);
```

Instead of using CmsAwaitResponses/CmsGetResponse pair, you can use this call alone. It turns the control over to the CMS system on processor 0, so it can be used as a worker. As soon as a response is available on processor 0, cms calls the user specified consumer function with two parameters: the response (a void *) and an integer refnum. (Question: should the size of the response be passed as a parameter to the consumer? User can do that as an explicit field of the response themselves, if necessary.)

```
void CmsExit(void);
```

Must be called on all processors to terminate execution.

Once either CmsProcessResponses or CmsAwaitResponses returns, you may fire the next batch of tasks via CmsFireTask again.

14.3.3 Example Program

```
#include "cms.h"

#define MAX 10

typedef struct {
    float a;
} Task;

typedef struct {
    float result;
} Response;

Task t;

int worker(Task *t, Response **r)
{
    /* do work and generate a single response */
    int i;
    Task *t1;
    int k;

    CmiPrintf("%d: in worker %f \n", CmiMyPe(), t->a);
    *r = (Response *) malloc(sizeof(Response));
    (*r)->result = t->a * t->a;
    return sizeof(Response);
}

int consumer(Response * r, int refnum)
```

(continues on next page)

(continued from previous page)

```

{
    CmiPrintf("consumer: response with refnum = %d is %f\n", refnum,
              r->result);
}

main(int argc, char *argv[])
{
    int i, j, k, ref;
    /* 2nd parameter is the max number of tasks
     * fired before "awaitResponses"
     */
    CmsInit((CmsWorkerFn)worker, 20);
    if (CmiMyPe() == 0) { /* I am the manager */
        CmiPrintf("manager initied\n");
        for (i = 0; i < 3; i++) { /* number of iterations or phases */
            /* prepare the next generation of problems to solve */
            /* then, fire the next batch of tasks for the worker */
            for (j = 0; j < 5; j++) {
                t.a = 10 * i + j;
                ref = j; /* a ref number to associate with the task, */
                /* so that the response for this task can be identified. */
                CmsFireTask(ref, &t, sizeof(t));
            }
            /* Now wait for the responses */
            CmsAwaitResponses(); /* allows proc 0 to be used as a worker. */
            /* Now extract the responses from the system */
            for (j = 0; j < 5; j++) {
                Response *r = (Response *) CmsGetResponse(j);
                CmiPrintf("Response %d is: %f \n", j, r->result);
            }
            /* End of one master-slave phase */
            CmiPrintf("End of phase %d\n", i);
        }

        CmiPrintf("Now the consumerFunction mode\n");

        if (CmiMyPe() == 0) { /* I am the manager */
            for (i = 0; i < 3; i++) {
                t.a = 5 + i;
                CmsFireTask(i, &t, sizeof(t));
            }
            CmsProcessResponses((CmsConsumerFn)consumer);
            /* Also allows proc. 0 to be used as a worker.
             * In addition, responses will be processed on processor 0
             * via the "consumer" function as soon as they are available
             */
        }
        CmsExit();
    }
}

```

14.4 Data Structures

In the course of developing Converse and Charm++ we had to implement a number of data structures efficiently. If the ANSI standard C library were available to us on all platforms, we could have used it, but that was not the case. Also,

we needed both the C and C++ bindings of most data structures. In most cases, the functionality we needed was also a subset of the C standard library functionality, and by avoiding virtual methods etc, we have tried to code the most efficient implementations of those data structures.

Since these data structures are already part of Converse and Charm++, they are available to the users of these system free of cost :-<). In this chapter we document the available functions.

14.4.1 Queues, Lists, FIFOs etc.

This data structure is based on circular buffer, and can be used both like a FIFO and a Stack.

The following functions are available for use in C:

```
typedef ... CdsFifo;
```

An opaque data type representing a queue of `void*` pointers.

```
CdsFifo CdsFifo_Create(void);
```

Creates a queue in memory and returns its pointer.

```
CdsFifo CdsFifo_Create_len(int len);
```

Creates a queue in memory with the initial buffer size of `len` entries and returns its pointer.

```
void CdsFifo_Enqueue(CdsFifo q, void *elt);
```

Appends `elt` at the end of `q`.

```
void *CdsFifo_Dequeue(CdsFifo q);
```

Removes an element from the front of the `q`, and returns it. Returns 0 if the queue is empty.

```
void *CdsFifo_Pop(CdsFifo q);
```

Removes an element from the front of the `q`, and returns it. Returns 0 if the queue is empty. An alias for the `dequeue` function.

```
void CdsFifo_Push(CdsFifo q, void *elt);
```

Inserts `elt` in the beginning of `q`.

```
int CdsFifo_Empty(CdsFifo q);
```

Returns 1 if the `q` is empty, 0 otherwise.

```
int CdsFifo_Length(CdsFifo q);
```

Returns the length of the `q`.

```
int CdsFifo_Peek(CdsFifo q);
```

Returns the element from the front of the `q` without removing it.

```
void CdsFifo_Destroy(CdsFifo q);
```

Releases memory used by `q`.

The following Templates are available for use in C++:

```
template<class T>
class CkQ {
    CkQ(); // default constructor
    CkQ(int initial_size); // constructor with initial buffer size
    ~CkQ(); // destructor
```

(continues on next page)

(continued from previous page)

```

int length(void); // returns length of the q
bool isEmpty(void); // returns true if q is empty, false otherwise
T deq(void); // removes and returns the front element
void enq(const T&); // appends at the end of the list
void push(const T&); // inserts in the beginning of the list
T& operator[] (size_t n); // returns the n'th element
};

```

14.5 Converse Pseudorandom Number Generator

Converse provides three different Linear Congruential Random Number Generators. Each random number stream has a cycle length of 2^{64} as opposed to ANSI C standard's 2^{48} . Also, each of the three random number streams can be split into a number of per processor streams, so that the random number sequences can be computed in parallel, and are reproducible. Furthermore, there is no implicit critical section in the random number generator, and yet, this functionality is thread-safe, because all the state information is stored in the structure allocated by the programmer. Further, this state information is stored in a first class object, and can be passed to other processors through messages. This module of Converse is based on the public-domain SPRNG¹ package developed by Ashok Srinivasan² at NCSA.

For minimal change to programs already using C functions `rand()`, `srand()`, and `drand48()`, Converse also maintains a “default” random number stream.

Interface to the Converse Pseudorandom Number Generator module is as follows:

```
typedef ... CrnStream;
```

State information for generating the next random number in the sequence.

```
void CrnInitStream(CrnStream *stream, int seed, int type)
```

Initializes the new random number stream `stream` of `type` using `seed`. `type` can have values 0, 1, or 2 to represent three types of linear congruential random number generators.

```
int CrnInt(CrnStream *stream)
```

Returns an integer between 0 and $2^{31} - 1$ corresponding to the next random number in the sequence associated with `stream`. Advances `stream` by one in the sequence.

```
double CrnDouble(CrnStream *stream)
```

Returns a double precision floating point number between 0 and 1 corresponding to the next random number in the sequence associated with `stream`. Advances `stream` by one in the sequence.

```
float CrnFloat(CrnStream *stream)
```

Returns a single precision floating point number between 0 and 1 corresponding to the next random number in the sequence associated with `stream`. Advances `stream` by one in the sequence.

```
void CrnSrand(int seed)
```

Specifies a different seed for the default random number stream. Replaces `srand()`.

```
int CrnRand(void)
```

Generate the next integer random number from the default random number stream. Replaces `rand()`.

```
double CrnDrand(void)
```

Generate the next double precision random number from the default random number stream. Replaces `drand48()`.

¹ URL: <http://www.ncsa.uiuc.edu/Apps/SPRNG/www/>

² Email: ashoks@ncsa.uiuc.edu

14.6 Automatic Parameter Marshalling

Automatic Parameter Marshalling is a concise means of invoking functions on remote processors. The CPM module handles all the details of packing, transmitting, translating, and unpacking the arguments. It also takes care of converting function pointers into handler numbers. With all these details out of the way, it is possible to perform remote function invocation in a single line of code.

14.6.1 CPM Basics

The heart of the CPM module is the CPM scanner. The scanner reads a C source file. When it sees the keyword `CpmInvokable` in front of one of the user's function declarations, it generates a *launcher* for that particular function. The *launcher* is a function whose name is `Cpm_` concatenated to the name of the user's function. The launcher accepts the same arguments as the user's function, plus a *destination* argument. Calling the *launcher* transmits a message to another processor determined by the *destination* argument. When the message arrives and is handled, the user's function is called.

For example, if the CPM scanner sees the following function declaration

```
CpmInvokable myfunc(int x, int y) { ... }
```

The scanner will generate a launcher named `Cpm_myfunc`. The launcher has this prototype:

```
void Cpm_myfunc(CpmDestination destination, int x, int y);
```

If one were to call `Cpm_myfunc` as follows:

```
Cpm_myfunc(CpmSend(3), 8, 9);
```

a message would be sent to processor 3 ordering it to call `myfunc(8, 9)`. Notice that the *destination* argument isn't just an integer processor number. The possible destinations for a message are described later.

When the CPM scanner is applied to a C source file with a particular name, it generates a certain amount of parameter packing and unpacking code, and this code is placed in an include file named similarly to the original C file: the `.c` is replaced with `.cpm.h`. The include file must be included in the original `.c` file, after the declarations of the types which are being packed and unpacked, but before all uses of the CPM invocation mechanisms.

Note that the `.cpm.h` include file is *not* for prototyping. It contains the C code for the packing and unpacking mechanisms. Therefore, it should only be included in the one source file from which it was generated. If the user wishes to prototype his code, he must do so normally, by writing a header file of his own.

Each `.cpm.h` file contains a function `CpmInitializeThisModule`, which initializes the code in *that* `.cpm.h` file. The function is declared `static`, so it is possible to have one in each `.cpm.h` file without conflicts. It is the responsibility of the CPM user to call each of these `CpmInitializeThisModule` functions before using any of the CPM mechanisms.

We demonstrate the use of the CPM mechanisms using the following short program `myprog.c`:

```
1  #include "myprog.cpm.h"
2
3  CpmInvokable print_integer(int n)
4  {
5      CmiPrintf("%d\n", n);
6  }
7
8  user_main(int argc, char **argv)
9  {
```

(continues on next page)

(continued from previous page)

```

10  int i;
11  CpmModuleInit();
12  CpmInitializeThisModule();
13  if (CmiMyPe()==0)
14      for (i=1; i<CmiNumPes(); i++)
15          Cpm_print_integer(CpmSend(i), rand());
16  }
17
18  main(int argc, char **argv)
19  {
20      ConverseInit(argc, argv, user_main, 0, 0);
21  }

```

Lines 3-6 of this program contain a simple C function that prints an integer. The function is marked with the word `CpmInvokable`. When the CPM scanner sees this word, it adds the function `Cpm_print_integer` to the file `myprog.cpm.h`. The program includes `myprog.cpm.h` on line 1, and initializes the code in there on line 12. Each call to `Cpm_print_integer` on line 15 builds a message that invokes `print_integer`. The destination-argument `CpmSend(i)` causes the message to be sent to the i 'th processor.

The effect of this program is that the first processor orders each of the other processors to print a random number. Note that the example is somewhat minimalist since it doesn't contain any code for terminating itself. Also note that it would have been more efficient to use an explicit broadcast. Broadcasts are described later.

All launchers accept a *CpmDestination* as their first argument. A *CpmDestination* is actually a pointer to a small C structure containing routing and handling information. The CPM module has many built-in functions that return *CpmDestinations*. Therefore, any of these can be used as the first argument to a launcher:

- **CpmSend(pe)** - the message is transmitted to processor *pe* with maximum priority.
- **CpmEnqueue(pe, queueing, priobits, prioptr)** - The message is transmitted to processor *pe*, where it is enqueued with the specified queueing strategy and priority. The *queueing*, *priobits*, and *prioptr* arguments are the same as for **CqsEnqueueGeneral**.
- **CpmEnqueueFIFO(pe)** - the message is transmitted to processor *pe* and enqueued with the middle priority (zero), and FIFO relative to messages with the same priority.
- **CpmEnqueueLIFO(pe)** - the message is transmitted to processor *pe* and enqueued with the middle priority (zero), and LIFO relative to messages with the same priority.
- **CpmEnqueueIFIFO(pe, prio)** - the message is transmitted to processor *pe* and enqueued with the specified integer-priority *prio*, and FIFO relative to messages with the same priority.
- **CpmEnqueueILIFO(pe, prio)** - the message is transmitted to processor *pe* and enqueued with the specified integer-priority *prio*, and LIFO relative to messages with the same priority.
- **CpmEnqueueBFIFO(pe, priobits, prioptr)** - the message is transmitted to processor *pe* and enqueued with the specified bitvector-priority, and FIFO relative to messages with the same priority.
- **CpmEnqueueBLIFO(pe, priobits, prioptr)** - the message is transmitted to processor *pe* and enqueued with the specified bitvector-priority, and LIFO relative to messages with the same priority.
- **CpmMakeThread(pe)** - The message is transmitted to processor *pe* where a *CthThread* is created, and the thread invokes the specified function.

All the functions shown above accept processor numbers as arguments. Instead of supplying a processor number, one can also supply the special symbols `CPM_ALL` or `CPM_OTHERS`, causing a broadcast. For example,

```
Cpm_print_integer(CpmMakeThread(CPM_ALL), 5);
```

would broadcast a message to all the processors causing each processor to create a thread, which would in turn invoke `print_integer` with the argument 5.

14.6.2 CPM Packing and Unpacking

Functions preceded by the word **CpmInvokable** must have simple argument lists. In particular, the argument list of a CpmInvokable function can only contain cpm-single-arguments and cpm-array-arguments, as defined by this grammar:

```
cpm-single-argument ::= typeword varname
cpm-array-argument  ::= typeword '*' varname
```

When CPM sees the cpm-array-argument notation, CPM interprets it as being a pointer to an array. In this case, CPM attempts to pack an entire array into the message, whereas it only attempts to pack a single element in the case of the cpm-single-argument notation.

Each cpm-array-argument must be preceded by a cpm-single-argument of type `CpmDim`. `CpmDim` is simply an alias for `int`, but when CPM sees an argument declared `CpmDim`, it knows that the next argument will be a cpm-array-argument, and it interprets the `CpmDim` argument to be the size of the array. Given a pointer to the array, its size, and its element-type, CPM handles the packing of array values as automatically as it handles single values.

A second program, `example2.c`, uses array arguments:

```
1  #include "example2.cpm.h"
2
3  CpmInvokable print_program_arguments(CpmDim argc, CpmStr *argv)
4  {
5      int i;
6      CmiPrintf("The program's arguments are: ");
7      for (i=0; i<argc; i++) CmiPrintf("%s ", argv[i]);
8      CmiPrintf("\n");
9  }
10
11 user_main(int argc, char **argv)
12 {
13     CpmModuleInit();
14     CpmInitializeThisModule();
15     if (CmiMyPe()==0)
16         Cpm_print_program_arguments(CpmSend(1), argc, argv);
17 }
18
19 main(int argc, char **argv)
20 {
21     ConverseInit(argc, argv, user_main, 0, 0);
22 }
```

The word `CpmStr` is a CPM built-in type, it represents a null-terminated string:

```
typedef char *CpmStr;
```

Therefore, the function `print_program_arguments` takes exactly the same arguments as `user_main`. In this example, the main program running on processor 0 transmits the arguments to processor 1, which prints them out.

Thus far, we have only shown functions whose prototypes contain builtin CPM types. CPM has built-in knowledge of the following types: `char`, `short`, `int`, `long`, `float`, `double`, `CpmDim`, and `CpmStr` (pointer to a null-terminated string). However, you may also transmit user-defined types in a CPM message.

For each (non-builtin) type the user wishes to pack, the user must supply some pack and unpack routines. The subroutines needed depend upon whether the type is a pointer or a simple type. Simple types are defined to be those that contain no pointers at all. Note that some types are neither pointers, nor simple types. CPM cannot currently handle such types.

CPM knows which type is which only through the following declarations:

```
CpmDeclareSimple(typeword);
CpmDeclarePointer(typeword);
```

The user must supply such declarations for each type that must be sent via CPM.

When packing a value v which is a simple type, CPM uses the following strategy. The generated code first converts v to network interchange format by calling `CpmPack_typename(&v)`, which must perform the conversion in-place. It then copies v byte-for-byte into the message and sends it. When the data arrives, it is extracted from the message and converted back using `CpmUnpack_typename(&v)`, again in-place. The user must supply the pack and unpack routines.

When packing a value v which is a pointer, the generated code determines how much space is needed in the message buffer by calling `CpmPtrSize_typename(v)`. It then transfers the data pointed to by v into the message using `CpmPtrPack_typename(p, v)`, where p is a pointer to the allocated space in the message buffer. When the message arrives, the generated code extracts the packed data from the message by calling `CpmPtrUnpack_typename(p)`. The unpack function must return a pointer to the unpacked data, which is allowed to still contain pointers to the message buffer (or simply be a pointer to the message buffer). When the invocation is done, the function `CpmPtrFree_typename(v)` is called to free any memory allocated by the unpack routine. The user must supply the size, pack, unpack, and free routines.

The following program fragment shows the declaration of two user-defined types:

```
1  typedef struct { double x,y; } coordinate;
2  CpmDeclareSimple(coordinate);
3
4  void CpmPack_coordinate(coordinate *p)
5  {
6      CpmPack_double(&(p->x));
7      CpmPack_double(&(p->y));
8  }
9
10 void CpmUnpack_coordinate(coordinate *p)
11 {
12     CpmUnpack_double(&(p->x));
13     CpmUnpack_double(&(p->y));
14 }
15
16 typedef int *intptr;
17 CpmDeclarePointer(intptr);
18
19 #define CpmPtrSize_intptr(p) sizeof(int)
20
21 void CpmPtrPack_intptr(void *p, intptr v)
22 {
23     *(int *)p = *v;
24     CpmPack_int((int *)p);
25 }
26
27 intptr CpmPtrUnpack_intptr(void *p)
28 {
29     CpmUnpack_int((int *)p);
```

(continues on next page)

(continued from previous page)

```

30     return (int *)p;
31 }
32
33 #define CpmPtrFree_intptr(p) (0)
34
35 #include "example3.cpm.h"
36 ...

```

The first type declared in this file is the coordinate. Line 2 contains the C type declaration, and line 3 notifies CPM that it is a simple type, containing no pointers. Lines 5-9 declare the pack function, which receives a pointer to a coordinate, and must pack it in place. It makes use of the pack-function for doubles, which also packs in place. The unpack function is similar.

The second type declared in this file is the `intptr`, which we intend to mean a pointer to a single integer. On line 18 we notify CPM that the type is a pointer, and that it should therefore use `CpmPtrSize_intptr`, `CpmPtrPack_intptr`, `CpmPtrUnpack_intptr`, and `CpmPtrFree_intptr`. Line 20 shows the size function, a constant: we always need just enough space to store one integer. The pack function copies the `int` into the message buffer, and packs it in place. The unpack function unpacks it in place, and returns an `intptr`, which points right to the unpacked integer which is still in the message buffer. Since the `int` is still in the message buffer, and not in dynamically allocated memory, the free function on line 34 doesn't have to do anything.

Note that the inclusion of the `.cpm.h` file comes after these type and pack declarations: the `.cpm.h` file will reference these functions and macros, therefore, they must already be defined.

14.6.3 Inventing New Types of CpmDestinations

It is possible for the user to create new types of `CpmDestinations`, and to write functions that return these new destinations. In order to do this, one must have a mental model of the steps performed when a Cpm message is sent. This knowledge is only necessary to those wishing to invent new kinds of destinations. Others can skip this section.

The basic steps taken when sending a CPM message are:

1. **The destination-structure is created.** The first argument to the launcher is a `CpmDestination`. Therefore, before the launcher is invoked, one typically calls a function (like `CpmSend`) to build the destination-structure.
2. **The launcher allocates a message-buffer.** The buffer contains space to hold a function-pointer and the function's arguments. It also contains space for an "envelope", the size of which is determined by a field in the destination-structure.
3. **The launcher stores the function-arguments in the message buffer.** In doing so, the launcher converts the arguments to a contiguous sequence of bytes.
4. **The launcher sets the message's handler.** For every launcher, there is a matching function called an *invoker*. The launcher's job is to put the argument data in the message and send the message. The *invoker*'s job is to extract the argument data from the message and call the user's function. The launcher uses `CmiSetHandler` to tell Converse to handle the message by calling the appropriate *invoker*.
5. **The message is sent, received, and handled.** The destination-structure contains a pointer to a send-function. The send-function is responsible for choosing the message's destination and making sure that it gets there and gets handled. The send-function has complete freedom to implement this in any manner it wishes. Eventually, though, the message should arrive at a destination and its handler should be called.
6. **The user's function is invoked.** The invoker extracts the function arguments from the message buffer and calls the user's function.

The *send-function* varies because messages take different routes to get to their final destinations. Compare, for example, `CpmSend` to `CpmEnqueueFIFO`. When `CpmSend` is used, the message goes straight to the target processor and

gets handled. When `CpmEnqueueFIFO` is used, the message goes to the target processor, goes into the queue, comes out of the queue, and *then* gets handled. The *send-function* must implement not only the transmission of the message, but also the possible “detouring” of the message through queues or into threads.

We now show an example CPM command, and describe the steps that are taken when the command is executed. The command we will consider is this one:

```
Cpm_print_integer(CpmEnqueueFIFO(3), 12);
```

Which sends a message to processor 3, ordering it to call `print_integer(12)`.

The first step is taken by `CpmEnqueueFIFO`, which builds the `CpmDestination`. The following is the code for `CpmEnqueueFIFO`:

```
typedef struct CpmDestinationSend_s
{
    void (*sendfn)();
    int envsize;
    int pe;
}
*CpmDestinationSend;

CpmDestination CpmEnqueueFIFO(int pe)
{
    static struct CpmDestinationSend_s ctrl;
    ctrl.envsize = sizeof(int);
    ctrl.sendfn = CpmEnqueueFIFO1;
    ctrl.pe = pe;
    return (CpmDestination)&ctrl;
}
```

Notice that the `CpmDestination` structure varies, depending upon which kind of destination is being used. In this case, the destination structure contains a pointer to the send-function `CpmEnqueueFIFO1`, a field that controls the size of the envelope, and the destination-processor. In a `CpmDestination`, the `sendfn` and `envsize` fields are required, additional fields are optional.

After `CpmEnqueueFIFO` builds the destination-structure, the launcher `Cpm_print_integer` is invoked. `Cpm_print_integer` performs all the steps normally taken by a launcher:

1. **It allocates the message buffer.** In this case, it sets aside just enough room for one int as an envelope, as dictated by the destination-structure’s `envsize` field.
2. **It stores the function-arguments in the message-buffer.** In this case, the function-arguments are just the integer 12.
3. **It sets the message’s handler.** In this case, the message’s handler is set to a function that will extract the arguments and call `print_integer`.
4. **It calls the send-function to send the message.**

The code for the send-function is here:

```
void *CpmEnqueueFIFO1(CpmDestinationSend dest, int len, void *msg)
{
    int *env = (int *)CpmEnv(msg);
    env[0] = CmiGetHandler(msg);
    CmiSetHandler(msg, CpvAccess(CpmEnqueueFIFO2_Index));
    CmiSyncSendAndFree(dest->pe, len, msg);
}
```

The send-function `CpmEnqueueFIFO1` starts by switching the handler. The original handler is removed using `CmiGetHandler`. It is set aside in the message buffer in the “envelope” space described earlier — notice the use of `CpmEnv` to obtain the envelope. This is the purpose of the envelope in the message — it is a place where the send-function can store information. The destination-function must anticipate how much space the send-function will need, and it must specify that amount of space in the destination-structure field *envsize*. In this case, the envelope is used to store the original handler, and the message’s handler is set to an internal function called `CpmEnqueueFIFO2`.

After switching the handler, `CpmEnqueueFIFO1` sends the message. Eventually, the message will be received by `CsdScheduler`, and its handler will be called. The result will be that `CpmEnqueueFIFO2` will be called on the destination processor. Here is the code for `CpmEnqueueFIFO2`:

```
void CpmEnqueueFIFO2(void *msg)
{
    int *env;
    CmiGrabBuffer(&msg);
    env = (int *)CpmEnv(msg);
    CmiSetHandler(msg, env[0]);
    CsdEnqueueFIFO(msg);
}
```

This function takes ownership of the message-buffer from `Converse` using `CmiGrabBuffer`. It extracts the original handler from the envelope (the handler that calls `print_integer`), and restores it using `CmiSetHandler`. Having done so, it enqueues the message with the FIFO queueing policy. Eventually, the scheduler picks the message from the queue, and `print_integer` is invoked.

In summary, the procedure for implementing new kinds of destinations is to write one send-function, one function returning a `CpmDestination` (which contains a reference to the send-function), and one or more `Converse` handlers to manipulate the message.

The destination-function must return a pointer to a “destination-structure”, which can in fact be any structure matching the following specifications:

- The first field must be a pointer to a send-function,
- The second field must be an integer, the envelope-size.

This pointer must be coerced to type `CpmDestination`.

The send-function must have the following prototype:

```
void sendfunction(CpmDestination dest, int msglen, void *msgptr)
```

It can access the envelope of the message using `CpmEnv`:

```
int *CpmEnv(void *msg);
```

It can also access the data stored in the destination-structure by the destination-function.

14.7 Load Balancing

14.7.1 Using Converse Load Balancers

This module defines a function **CldEnqueue** that sends a message to a lightly-loaded processor. It automates the process of finding a lightly-loaded processor.

The function **CldEnqueue** is extremely sophisticated. It does not choose a processor, send the message, and forget it. Rather, it puts the message into a pool of movable work. The pool of movable work gradually shrinks as it is consumed

(processed), but in most programs, there is usually quite a bit of movable work available at any given time. As load conditions shift, the load balancers shifts the pool around, compensating. Any given message may be shifted more than once, as part of the pool.

CldEnqueue also accounts for priorities. Normal load-balancers try to make sure that all processors have some work to do. The function **CldEnqueue** goes a step further: it tries to make sure that all processors have some reasonably high-priority work to do. This can be extremely helpful in AI search applications.

The two assertions above should be qualified: **CldEnqueue** can use these sophisticated strategies, but it is also possible to configure it for different behavior. When you compile and link your program, you choose a *load-balancing strategy*. That means you link in one of several implementations of the load-balancer. Most are sophisticated, as described above. But some are simple and cheap, like the random strategy. The process of choosing a strategy is described in the manual *Converse Installation and Usage*.

Before you send a message using **CldEnqueue**, you must write an *info* function with this prototype:

```
void InfoFn(void *msg, CldPackFn *pfn, int *len, int *queueing, int *priobits,
unsigned int *prioPtr);
```

The load balancer will call the info function when it needs to know various things about the message. The load balancer will pass in the message via the parameter *msg*. The info function's job is to "fill in" the other parameters. It must compute the length of the message, and store it at **len*. It must determine the *pack* function for the message, and store a pointer to it at **pfn*. It must identify the priority of the message, and the queueing strategy that must be used, storing this information at **queueing*, **priobits*, and **prioPtr*. Caution: the priority will not be copied, so the **prioPtr* should probably be made to point to the message itself.

After the user of **CldEnqueue** writes the "info" function, the user must register it, using this:

```
int CldRegisterInfoFn(CldInfoFn fn)
```

Accepts a pointer to an info-function. Returns an integer index for the info-function. This index will be needed in **CldEnqueue**.

Normally, when you send a message, you pack up a bunch of data into a message, send it, and unpack it at the receiving end. It is sometimes possible to perform an optimization, though. If the message is bound for a processor within the same address space, it isn't always necessary to copy all the data into the message. Instead, it may be sufficient to send a message containing only a pointer to the data. This saves much packing, unpacking, and copying effort. It is frequently useful, since in a properly load-balanced program, a great many messages stay inside a single address space.

With **CldEnqueue**, you don't know in advance whether a message is going to cross address-space boundaries or not. If it's to cross address spaces, you need to use the "long form", but if it's to stay inside an address space, you want to use the faster "short form". We call this "conditional packing." When you send a message with **CldEnqueue**, you should initially assume it will not cross address space boundaries. In other words, you should send the "short form" of the message, containing pointers. If the message is about to leave the address space, the load balancer will call your pack function, which must have this prototype:

```
void PackFn(void **msg)
```

The pack function is handed a pointer to a pointer to the message (yes, a pointer to a pointer). The pack function is allowed to alter the message in place, or replace the message with a completely different message. The intent is that the pack function should replace the "short form" of the message with the "long form" of the message. Note that if it replaces the message, it should *CmiFree* the old message.

Of course, sometimes you don't use conditional packing. In that case, there is only one form of the message. In that case, your pack function can be a no-op.

Pack functions must be registered using this:

```
int CldRegisterPackFn(CldPackFn fn)
```

Accepts a pointer to an pack-function. Returns an integer index for the pack-function. This index will be needed in **CldEnqueue**.

Normally, **CldEnqueue** sends a message to a lightly-loaded processor. After doing this, it enqueues the message with the appropriate priority. The function **CldEnqueue** can also be used as a mechanism to simply enqueue a message on a remote processor with a priority. In other words, it can be used as a prioritized send-function. To do this, one of the **CldEnqueue** parameters allows you to override the load-balancing behavior and lets you choose a processor yourself.

The prototype for **CldEnqueue** is as follows:

```
void CldEnqueue(int pe, void *msg, int infofn)
```

The argument `msg` is a pointer to the message. The parameter `infofn` represents a function that can analyze the message. The parameter `packfn` represents a function that can pack the message. If the parameter `pe` is `CLD_ANYWHERE`, the message is sent to a lightly-loaded processor and enqueued with the appropriate priority. If the parameter `pe` is a processor number, the message is sent to the specified processor and enqueued with the appropriate priority. **CldEnqueue** frees the message buffer using **CmiFree**.

The following simple example illustrates how a Converse program can make use of the load balancers.

hello.c:

```
#include <stdio.h>
#include "converse.h"
#define CHARES 10

void startup(int argc, char *argv[]);
void registerAndInitialize();

typedef struct pmsgstruct
{
    char header[CmiExtHeaderSizeBytes];
    int pe, id, pfnx;
    int queuing, priobits;
    unsigned int prioptr;
} pmsg;

CpvDeclare(int, MyHandlerIndex);
CpvDeclare(int, InfoFnIndex);
CpvDeclare(int, PackFnIndex);

int main(int argc, char *argv[])
{
    ConverseInit(argc, argv, startup, 0, 0);
    CsdScheduler(-1);
}

void startup(int argc, char *argv[])
{
    pmsg *msg;
    int i;

    registerAndInitialize();
    for (i=0; i<CHARES; i++) {
        msg = (pmsg *)malloc(sizeof(pmsg));
        msg->pe = CmiMyPe();
        msg->id = i;
        msg->pfnx = CpvAccess(PackFnIndex);
        msg->queuing = CQS_QUEUEING_FIFO;
```

(continues on next page)

(continued from previous page)

```

    msg->priobits = 0;
    msg->prioPtr = 0;
    CmiSetHandler(msg, CpvAccess(MyHandlerIndex));
    CmiPrintf("[%d] sending message %d\n", msg->pe, msg->id);
    CldEnqueue(CLD_ANYWHERE, msg, CpvAccess(InfoFnIndex));
    /*    CmiSyncSend(i, sizeof(pemsg), &msg); */
}
}

void MyHandler(pemsg *msg)
{
    CmiPrintf("Message %d created on %d handled by %d.\n", msg->id, msg->pe,
        CmiMyPe());
}

void InfoFn(pemsg *msg, CldPackFn *pfn, int *len, int *queuing, int *priobits,
    unsigned int *prioPtr)
{
    *pfn = (CldPackFn)CmiHandlerToFunction(msg->pfnx);
    *len = sizeof(pemsg);
    *queuing = msg->queuing;
    *priobits = msg->priobits;
    prioPtr = &(msg->prioPtr);
}

void PackFn(pemsg **msg)
{
}

void registerAndInitialize()
{
    CpvInitialize(int, MyHandlerIndex);
    CpvAccess(MyHandlerIndex) = CmiRegisterHandler(MyHandler);
    CpvInitialize(int, InfoFnIndex);
    CpvAccess(InfoFnIndex) = CldRegisterInfoFn((CldInfoFn)InfoFn);
    CpvInitialize(int, PackFnIndex);
    CpvAccess(PackFnIndex) = CldRegisterPackFn((CldPackFn)PackFn);
}

```

14.7.2 How to Write a Load Balancer for Converse/Charm++

Introduction

This manual details how to write your own general-purpose message-based load balancer for Converse. A Converse load balancer can be used by any Converse program, but also serves as a *seed* load balancer for Charm++ chare creation messages. Specifically, to use a load balancer, you would pass messages to `CldEnqueue` rather than directly to the scheduler. This is the default behavior with chare creation message in Charm++. Thus, the primary provision of a new load balancer is an implementation of the `CldEnqueue` function.

Existing Load Balancers and Provided Utilities

Throughout this manual, we will occasionally refer to the source code of two provided load balancers, the random initial placement load balancer (`cldb.rand.c`) and the virtual topology-based load balancer (`cldb.neighbor`).

c) which applies virtual topology including dense graph to construct neighbors. The functioning of these balancers is described in the Charm++ manual load balancing section.

In addition, a special utility is provided that allows us to add and remove load-balanced messages from the scheduler's queue. The source code for this is available in `clldb.c`. The usage of this utility will also be described here in detail.

14.7.3 A Sample Load Balancer

This manual steps through the design of a load balancer using an example which we will call `test`. The `test` load balancer has each processor periodically send half of its load to its neighbor in a ring. Specifically, for N processors, processor K will send approximately half of its load to $(K+1)\%N$, every 100 milliseconds (this is an example only; we leave the genius approaches up to you).

Minimal Requirements

The minimal requirements for a load balancer are illustrated by the following code.

```
#include <stdio.h>
#include "converse.h"

const char *CldGetStrategy(void)
{
    return "test";
}

CpvDeclare(int, CldHandlerIndex);

void CldHandler(void *msg)
{
    CldInfoFn ifn; CldPackFn pfn;
    int len, queueing, priobits; unsigned int *prioptr;

    CmiGrabBuffer((void **) &msg);
    CldRestoreHandler(msg);
    ifn = (CldInfoFn) CmiHandlerToFunction(CmiGetInfo(msg));
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
}

void CldEnqueue(int pe, void *msg, int infofn)
{
    int len, queueing, priobits; unsigned int *prioptr;
    CldInfoFn ifn = (CldInfoFn) CmiHandlerToFunction(infofn);
    CldPackFn pfn;

    if (pe == CLD_ANYWHERE) {
        /* do what you want with the message; in this case we'll just keep
           it local */
        ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
        CmiSetInfo(msg, infofn);
        CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
    }
    else {
        /* pe contains a particular destination or broadcast */
        ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    }
}
```

(continues on next page)

(continued from previous page)

```

    if (pfn) {
        pfn(&msg);
        ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    }
    CldSwitchHandler(msg, CpvAccess(CldHandlerIndex));
    CmiSetInfo(msg, infofn);
    if (pe==CLD_BROADCAST)
        CmiSyncBroadcastAndFree(len, msg);
    else if (pe==CLD_BROADCAST_ALL)
        CmiSyncBroadcastAllAndFree(len, msg);
    else CmiSyncSendAndFree(pe, len, msg);
}
}

void CldModuleInit()
{
    char *argv[] = { NULL };
    CpvInitialize(int, CldHandlerIndex);
    CpvAccess(CldHandlerIndex) = CmiRegisterHandler(CldHandler);
    CldModuleGeneralInit(argv);
}

```

The primary function a load balancer must provide is the **CldEnqueue** function, which has the following prototype:

```
void CldEnqueue(int pe, void *msg, int infofn);
```

This function takes three parameters: *pe*, *msg* and *infofn*. *pe* is the intended destination of the *msg*. *pe* may take on one of the following values:

- Any valid processor number - the message must be sent to that processor
- CLD_ANYWHERE - the message can be placed on any processor
- CLD_BROADCAST - the message must be sent to all processors excluding the local processor
- CLD_BROADCAST_ALL - the message must be sent to all processors including the local processor

CldEnqueue must handle all of these possibilities. The only case in which the load balancer should get control of a message is when *pe* = CLD_ANYWHERE. All other messages must be sent off to their intended destinations and passed on to the scheduler as if they never came in contact with the load balancer.

The integer parameter *infofn* is a handler index for a user-provided function that allows **CldEnqueue** to extract information about (mostly components of) the message *msg*.

Thus, an implementation of the **CldEnqueue** function might have the following structure:

```

void CldEnqueue(int pe, void *msg, int infofn)
{
    ...
    if (pe == CLD_ANYWHERE)
        /* These messages can be load balanced */
    else if (pe == CmiMyPe())
        /* Enqueue the message in the scheduler locally */
    else if (pe==CLD_BROADCAST)
        /* Broadcast to all but self */
    else if (pe==CLD_BROADCAST_ALL)
        /* Broadcast to all plus self */
    else /* Specific processor number was specified */
        /* Send to specific processor */
}

```

In order to fill in the code above, we need to know more about the message before we can send it off to a scheduler's queue, either locally or remotely. For this, we have the info function. The prototype of an info function must be as follows:

```
void ifn(void *msg, CldPackFn *pfn, int *len, int *queueing, int *priobits,
unsigned int **prioPtr);
```

Thus, to use the info function, we need to get the actual function via the handler index provided to **CldEnqueue**. Typically, **CldEnqueue** would contain the following declarations:

```
int len, queueing, priobits;
unsigned int *prioPtr;
CldPackFn pfn;
CldInfoFn ifn = (CldInfoFn)CmiHandlerToFunction(infofn);
```

Subsequently, a call to ifn would look like this:

```
ifn(msg, &pfn, &len, &queueing, &priobits, &prioPtr);
```

The info function extracts information from the message about its size, queuing strategy and priority, and also a pack function, which will be used when we need to send the message elsewhere. For now, consider the case where the message is to be locally enqueued:

```
...
else if (pe == CmiMyPe())
{
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioPtr);
    CsdEnqueueGeneral(msg, queueing, priobits, prioPtr);
}
...
```

Thus, we see the info function is used to extract info from the message that is necessary to pass on to **CsdEnqueueGeneral**.

In order to send the message to a remote destination and enqueue it in the scheduler, we need to pack it up with a special pack function so that it has room for extra handler information and a reference to the info function. Therefore, before we handle the last three cases of **CldEnqueue**, we have a little extra work to do:

```
...
else
{
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioPtr);
    if (pfn) {
        pfn(&msg);
        ifn(msg, &pfn, &len, &queueing, &priobits, &prioPtr);
    }
    CldSwitchHandler(msg, CpvAccess(CldHandlerIndex));
    CmiSetInfo(msg, infofn);
    ...
}
```

Calling the info function once gets the pack function we need, if there is one. We then call the pack function which rearranges the message leaving space for the info function, which we will need to call on the message when it is received at its destination, and also room for the extra handler that will be used on the receiving side to do the actual enqueueing. **CldSwitchHandler** is used to set this extra handler, and the receiving side must restore the original handler.

In the above code, we call the info function again because some of the values may have changed in the packing process.

Finally, we handle our last few cases:

```

...
    if (pe==CLD_BROADCAST)
        CmiSyncBroadcastAndFree(len, msg);
    else if (pe==CLD_BROADCAST_ALL)
        CmiSyncBroadcastAllAndFree(len, msg);
    else CmiSyncSendAndFree(pe, len, msg);
}
}

```

The above example also provides **CldHandler** which is used to receive messages that **CldEnqueue** forwards to other processors.

```

CpvDeclare(int, CldHandlerIndex);

void CldHandler(void *msg)
{
    CldInfoFn ifn; CldPackFn pfn;
    int len, queueing, priobits; unsigned int *prioPtr;

    CmiGrabBuffer((void **) &msg);
    CldRestoreHandler(msg);
    ifn = (CldInfoFn) CmiHandlerToFunction(CmiGetInfo(msg));
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioPtr);
    CsdEnqueueGeneral(msg, queueing, priobits, prioPtr);
}

```

Note that the **CldHandler** properly restores the message's original handler using **CldRestoreHandler**, and calls the info function to obtain the proper parameters to pass on to the scheduler. We talk about this more below.

Finally, Converse initialization functions call **CldModuleInit** to initialize the load balancer module.

```

void CldModuleInit()
{
    char *argv[] = { NULL };
    CpvInitialize(int, CldHandlerIndex);
    CpvAccess(CldHandlerIndex) = CmiRegisterHandler(CldHandler);
    CldModuleGeneralInit(argv);

    /* call other init processes here */
    CldBalance();
}

```

Provided Load Balancing Facilities

Converse provides a number of structures and functions to aid in load balancing (see `cldb.c`). Foremost amongst these is a method for queuing tokens of messages in a processor's scheduler in a way that they can be removed and relocated to a different processor at any time. The interface for this module is as follows:

```

void CldSwitchHandler(char *cmsg, int handler)
void CldRestoreHandler(char *cmsg)
int CldCountTokens()
int CldLoad()
void CldPutToken(char *msg)
void CldGetToken(char **msg)
void CldModuleGeneralInit(char **argv)

```

Messages normally have a handler index associated with them, but in addition they have extra space for an additional handler. This is used by the load balancer when we use an intermediate handler (typically **CldHandler**) to handle the message when it is received after relocation. To do this, we use **CldSwitchHandler** to temporarily swap the intended handler with the load balancer handler. When the message is received, **CldRestoreHandler** is used to change back to the intended handler.

CldPutToken puts a message in the scheduler queue in such a way that it can be retrieved from the queue. Once the message gets handled, it can no longer be retrieved. **CldGetToken** retrieves a message that was placed in the scheduler queue in this way. **CldCountTokens** tells you how many tokens are currently retrievable. **CldLoad** gives a slightly more accurate estimate of message load by counting the total number of messages in the scheduler queue.

CldModuleGeneralInit is used to initialize this load balancer helper module. It is typically called from the load balancer's **CldModuleInit** function.

The helper module also provides the following functions:

```
void CldMultipleSend(int pe, int numToSend)
int CldRegisterInfoFn(CldInfoFn fn)
int CldRegisterPackFn(CldPackFn fn)
```

CldMultipleSend is generally useful for any load balancer that sends multiple messages to one processor. It works with the token queue module described above. It attempts to retrieve up to `numToSend` messages, and then packs them together and sends them, via `CmiMultipleSend`, to `pe`. If the number and/or size of the messages sent is very large, **CldMultipleSend** will transmit them in reasonably sized parcels. In addition, the **CldBalanceHandler** and its associated declarations and initializations are required to use it.

You may want to use the three status variables. These can be used to keep track of what your LB is doing (see usage in `cldb.neighbor.c` and `itc++queens` program).

```
CpvDeclare(int, CldRelocatedMessages);
CpvDeclare(int, CldLoadBalanceMessages);
CpvDeclare(int, CldMessageChunks);
```

The two register functions register *info* and *pack* functions, returning an index for the functions. Info functions are used by the load balancer to extract the various components from a message. Amongst these components is the pack function index. If necessary, the pack function can be used to pack a message that is about to be relocated to another processor. Information on how to write info and pack functions is available in the load balancing section of the Converse Extensions manual.

Finishing the Test Balancer

The `test` balancer is a somewhat silly strategy in which every processor attempts to get rid of half of its load by periodically sending it to someone else, regardless of the load at the destination. Hopefully, you won't actually use this for anything important!

The `test` load balancer is available in `charm/src/Common/conv-ldb/cldb.test.c`. To try out your own load balancer you can use this filename and `SUPER_INSTALL` will compile it and you can link it into your Charm++ programs with `-balance test`. (To add your own new balancers permanently and give them another name other than "test" you will need to change the Makefile used by `SUPER_INSTALL`. Don't worry about this for now.) The `cldb.test.c` provides a good starting point for new load balancers.

Look at the code for the `test` balancer below, starting with the **CldEnqueue** function. This is almost exactly as described earlier. One exception is the handling of a few extra cases: specifically if we are running the program on only one processor, we don't want to do any load balancing. The other obvious difference is in the first case: how do we handle messages that can be load balanced? Rather than enqueueing the message directly with the scheduler, we make use of the token queue. This means that messages can later be removed for relocation. **CldPutToken** adds the message to the token queue on the local processor.

```

#include <stdio.h>
#include "converse.h"
#define PERIOD 100
#define MAXMSGBFRSIZE 100000

const char *CldGetStrategy(void)
{
    return "test";
}

CpvDeclare(int, CldHandlerIndex);
CpvDeclare(int, CldBalanceHandlerIndex);
CpvDeclare(int, CldRelocatedMessages);
CpvDeclare(int, CldLoadBalanceMessages);
CpvDeclare(int, CldMessageChunks);

void CldDistributeTokens()
{
    int destPe = (CmiMyPe()+1)%CmiNumPes(), numToSend;

    numToSend = CldLoad() / 2;
    if (numToSend > CldCountTokens())
        numToSend = CldCountTokens() / 2;
    if (numToSend > 0)
        CldMultipleSend(destPe, numToSend);
    CcdCallFnAfter((CcdVoidFn)CldDistributeTokens, NULL, PERIOD);
}

void CldBalanceHandler(void *msg)
{
    CmiGrabBuffer((void **) &msg);
    CldRestoreHandler(msg);
    CldPutToken(msg);
}

void CldHandler(void *msg)
{
    CldInfoFn ifn; CldPackFn pfn;
    int len, queueing, priobits; unsigned int *prioptr;

    CmiGrabBuffer((void **) &msg);
    CldRestoreHandler(msg);
    ifn = (CldInfoFn)CmiHandlerToFunction(CmiGetInfo(msg));
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
}

void CldEnqueue(int pe, void *msg, int infofn)
{
    int len, queueing, priobits; unsigned int *prioptr;
    CldInfoFn ifn = (CldInfoFn)CmiHandlerToFunction(infofn);
    CldPackFn pfn;

    if ((pe == CLD_ANYWHERE) && (CmiNumPes() > 1)) {
        ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
        CmiSetInfo(msg, infofn);
        CldPutToken(msg);
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
    else if ((pe == CmiMyPe()) || (CmiNumPes() == 1)) {
        ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
        CmiSetInfo(msg, infofn);
        CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
    }
    else {
        ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
        if (pfn) {
            pfn(&msg);
            ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
        }
        CldSwitchHandler(msg, CpvAccess(CldHandlerIndex));
        CmiSetInfo(msg, infofn);
        if (pe == CLD_BROADCAST)
            CmiSyncBroadcastAndFree(len, msg);
        else if (pe == CLD_BROADCAST_ALL)
            CmiSyncBroadcastAllAndFree(len, msg);
        else CmiSyncSendAndFree(pe, len, msg);
    }
}

void CldModuleInit()
{
    char *argv[] = { NULL };
    CpvInitialize(int, CldHandlerIndex);
    CpvAccess(CldHandlerIndex) = CmiRegisterHandler(CldHandler);
    CpvInitialize(int, CldBalanceHandlerIndex);
    CpvAccess(CldBalanceHandlerIndex) = CmiRegisterHandler(CldBalanceHandler);
    CpvInitialize(int, CldRelocatedMessages);
    CpvInitialize(int, CldLoadBalanceMessages);
    CpvInitialize(int, CldMessageChunks);
    CpvAccess(CldRelocatedMessages) = CpvAccess(CldLoadBalanceMessages) =
        CpvAccess(CldMessageChunks) = 0;
    CldModuleGeneralInit(argv);
    if (CmiNumPes() > 1)
        CldDistributeTokens();
}

```

Now look two functions up from **CldEnqueue**. We have an additional handler besides the **CldHandler**: the **CldBalanceHandler**. The purpose of this special handler is to receive messages that can be still be relocated again in the future. Just like the first case of **CldEnqueue** uses **CldPutToken** to keep the message retrievable, **CldBalanceHandler** does the same with relocatable messages it receives. **CldHandler** is only used when we no longer want the message to have the potential for relocation. It places messages irretrievably in the scheduler queue.

Next we look at our initialization functions to see how the process gets started. The **CldModuleInit** function gets called by the common Converse initialization code and starts off the periodic load distribution process by making a call to **CldDistributeTokens**. The entirety of the balancing is handled by the periodic invocation of this function. It computes an approximation of half of the PE's total load (**CsdLength()**), and if that amount exceeds the number of movable messages (**CldCountTokens()**), we attempt to move all of the movable messages. To do this, we pass this number of messages to move and the number of the PE to move them to, to the **CldMultipleSend** function.

14.8 Futures

This library supports the *future* abstraction, defined and used by Halstead and other researchers.

Cfuture CfutureCreate()

Returns the handle of an empty future. The future is said to reside on the processor that created it. The handle is a *global* reference to the future, in other words, it may be copied freely across processors. However, while the handle may be moved across processors freely, some operations can only be performed on the processor where the future resides.

Cfuture CfutureSet(Cfuture future, void *value, int nbytes)

Makes a copy of the value and stores it in the future. CfutureSet may be performed on processors other than the one where the future resides. If done remotely, the copy of the value is created on the processor where the future resides.

void *CfutureWait(Cfuture fut)

Waits until the future has been filled, then returns a pointer to the contents of the future. If the future has already been filled, this happens immediately (without blocking). Caution: CfutureWait can only be done on the processor where the Cfuture resides. A second caution: blocking operations (such as this one) can only be done in user-created threads.

void CfutureDestroy(Cfuture f)

Frees the space used by the specified Cfuture. This also frees the value stored in the future. Caution: this operation can only be done on the processor where the Cfuture resides.

void* CfutureCreateValue(int nbytes)

Allocates the specified amount of memory and returns a pointer to it. This buffer can be filled with data and stored into a future, using CfutureStoreBuffer below. This combination is faster than using CfutureSet directly.

void CfutureStoreValue(Cfuture fut, void *value)

Make a copy of the value and stores it in the future, destroying the original copy of the value. This may be significantly faster than the more general function, CfutureSet (it may avoid copying). This function can *only* be used to store values that were previously extracted from other futures, or values that were allocated using CfutureCreateValue.

void CfutureModuleInit()

This function initializes the futures module. It must be called once on each processor, during the handler-registration process (see the Converse manual regarding CmiRegisterHandler).

14.9 Converse-POSIX threads

We have implemented the POSIX threads API on top of Converse threads. To use the Converse-pthreads, you must include the header file:

```
#include <cpthreads.h>
```

Refer to the POSIX threads documentation for the documentation on the pthreads functions and types. Although Converse-pthreads threads are POSIX-compliant in most ways, there are some specific things one needs to know to use our implementation.

14.9.1 Pthreads and Converse

Our pthreads implementation is designed to exist within a Converse environment. For example, to send messages inside a POSIX program, you would still use the usual Converse messaging primitives.

14.9.2 Suppressing Name Conflicts

Some people may wish to use Converse pthreads on machines that already have a pthreads implementation in the standard library. This may cause some name-conflicts as we define the pthreads functions, and the system include files do too. To avoid such conflicts, we provide an alternative set of names beginning with the word Cpthread. These names are interchangeable with their pthread equivalents. In addition, you may prevent Converse from defining the pthread names at all with the preprocessor symbol `SUPPRESS_PTHREADS`:

```
#define SUPPRESS_PTHREADS
#include <cpthread.h>
```

14.9.3 Interoperating with Other Thread Packages

Converse programs are typically multilingual programs. There may be modules written using POSIX threads, but other modules may use other thread APIs. The POSIX threads implementation has the following restriction: you may only call the pthreads functions from inside threads created with `pthread_create`. Threads created by other thread packages (for example, the CthThread package) may not use the pthreads functions.

14.9.4 Preemptive Context Switching

Most implementations of POSIX threads perform time-slicing: when a thread has run for a while, it automatically gives up the CPU to another thread. Our implementation is currently nonpreemptive (no time-slicing). Threads give up control at two points:

- If they block (eg, at a mutex).
- If they call `pthread_yield()`.

Usually, the first rule is sufficient to make most programs work. However, a few programs (particularly, those that busy-wait) may need explicit insertion of yields.

14.9.5 Limits on Blocking Operations in main

Converse has a rule about blocking operations — there are certain pieces of code that may not block. This was an efficiency decision. In particular, the main function, Converse handlers, and the Converse startup function (see `ConverseInit`) may not block. You must be aware of this when using the POSIX threads functions with Converse.

There is a contradiction here — the POSIX standard requires that the pthreads functions work from inside `main`. However, many of them block, and Converse forbids blocking inside `main`. This contradiction can be resolved by renaming your posix-compliant `main` to something else: for example, `mymain`. Then, through the normal Converse startup procedure, create a POSIX thread to run `mymain`. We provide a convenience function to do this, called `Cpthread_start_main`. The startup code will be much like this:

```
void mystartup(int argc, char **argv)
{
    CpthreadModuleInit();
    Cpthread_start_main(mymain, argc, argv);
}

int main(int argc, char **argv)
{
    ConverseInit(mystartup, argc, argv, 0, 0);
}
```


This creates the first POSIX thread on each processor, which runs the function `mymain`. The `mymain` function is executing in a POSIX thread, and it may use any pthread function it wishes.

14.9.6 CpthreadModuleInit

On each processor, the function `CpthreadModuleInit` must be called before any other pthread function is called. This is shown in the example in the previous section.

14.10 Parallel Arrays of Threads

This module is `CPath`: Converse Parallel Array of Threads. It makes it simple to create arrays of threads, where the threads are distributed across the processors. It provides simple operations like sending a message to a thread, as well as group operations like multicasting to a row of threads, or reducing over an array of threads.

14.10.1 Creating Arrays of Threads

This module defines a data type `CPath`, also known as an “array descriptor”. Arrays are created by the function `CPathMakeArray`, and individual threads are created using `CPathMakeThread`:

```
void CPathMakeArray(CPath *path, int threadfn, int mapfn, ...)
```

This function initiates the creation of an array of threads. It fills in the array descriptor `*path`. Each thread in the array starts executing the function represented by `threadfn`. The function `mapfn` represents a mapping function, controlling the layout of the array. This parameter must be followed by the dimensions of the array, and then a zero.

```
void CPathMakeThread(CPath *path, int startfn, int pe)
```

This function makes a zero-dimensional array of threads, in other words, just one thread.

14.10.2 Mapping Functions for Arrays of Threads

One of the parameters to `CPathMakeArray` is a “mapping function”, which maps array elements to processors. Mapping functions must be registered. The integer index returned by the registration process is the number which is passed to `CPathMakeArray`. Mapping functions receive the array descriptor as a parameter, and may use it to determine the dimensions of the array.

```
unsigned int MapFn(CPath *path, int *indices)
```

This is a prototype map function, all mapping functions must have this parameter list. It accepts an array descriptor and a set of indices. It returns the processor number of the specified element.

```
int CPathRegisterMapper(void *mapfn)
```

Accepts a pointer to a mapping function, and returns an integer index for the function. This number can be used as a parameter to `CPathMakeArray`.

```
int CPathArrayDimensions(CPath *path)
```

Returns the number of dimensions in the specified array.

```
int CPathArrayDimension(CPath *path, int n)
```

Returns the `n`th dimension of the specified array.

14.10.3 Thread Functions for Arrays of Threads

Thread functions (the functions that the threads execute) must have the following prototype, and must be registered using the following registration function. The integer index returned by the registration process is the number which is passed to `CPathMakeArray`.

```
void ThreadFn(CPath *self, int *indices)
```

This is a prototype thread function. All thread-functions must have these parameters. When an array of threads is created, each thread starts executing the specified thread function. The function receives a pointer to a copy of the array's descriptor, and the array element's indices.

```
int CPathRegisterThreadFn(void *mapfn)
```

Accepts a pointer to a thread function, and returns an integer index for the function. This number can be used as a parameter to `CPathMakeArray`.

14.10.4 Sending Messages to Threads

Threads may send messages to each other using `CPathSend`, which takes a complicated set of parameters. The parameters are most easily described by a context-free grammar:

```
void CPathSend(dest-clause, tag-clause, data-clause, end-clause)
```

Where:

```
dest-clause ::= CPATH_DEST ',' pathptr ',' index ',' index ',' ...
tag-clause  ::= CPATH_TAG ',' tag
tag-clause  ::= CPATH_TAGS ',' tag ',' tag ',' ... ',' 0
tag-clause  ::= CPATH_TAGVEC ',' numtags ',' tagvector
data-clause ::= CPATH_BYTES ',' numbytes ',' bufptr
end-clause  ::= CPATH_END
```

The symbols `CPATH_DEST`, `CPATH_TAG`, `CPATH_TAGS`, `CPATH_TAGVEC`, `CPATH_BYTES`, `CPATH_END`, and the comma are terminal symbols. The symbols `descriptor`, `index`, `tag`, `numtags`, `tagvector`, `numbytes`, and `bufptr` all represent C expressions.

The `dest-clause` specifies which array and which indices the message is to go to. One must provide a pointer to an array descriptor and a set of indices. Any index may be either a normal index, or the wildcard `CPATH_ALL`. Using the wildcard causes a multicast. The `tag-clause` provides several notations, all of which specify an array of one or more integer tags to be sent with the message. These tags can be used at the receiving end for pattern matching. The `data-clause` specifies the data to go in the message, as a sequence of bytes. The `end-clause` represents the end of the parameter list.

Messages sent with `CPathSend` can be received using `CPathRecv`, analyzed using `CPathMsgDecodeBytes`, and finally discarded with `CPathMsgFree`:

```
void *CPathRecv(tag-clause, end-clause)
```

The `tag-clause` and `end-clause` match the grammar for `CPathSend`. The function will wait until a message with the same tags shows up (it waits using the thread-blocking primitives, see *Converse threads*). If any position in the `CPathRecv` tag-vector is `CPATH_WILD`, then that one position is ignored. `CPathRecv` returns an “opaque `CPath` message”. The message contains the data somewhere inside it. The data can be located using `CPathMsgDecodeBytes`, below. The opaque `CPath` message can be freed using `CPathMsgFree` below.

```
void CPathMsgDecodeBytes(void *msg, int *len, void *bytes)
```

Given an opaque `CPath` message (as sent by `CPathSend` and returned by `CPathRecv`), this function will locate the data inside it. The parameter `*len` is filled in with the data length, and `*bytes` is filled in with a pointer to the data bytes. Bear in mind that once you free the opaque `CPath` message, this pointer is no longer valid.

```
void CPathMsgFree(void *msg)
```

Frees an opaque CPath message.

14.10.5 Performing Reductions over Array Elements

An set of threads may participate in a reduction. All the threads wishing to participate must call CPathReduce. The parameters to CPathReduce are most easily described by a context-free grammar:

```
void CPathReduce(over-clause, tag-clause, red-clause, data-clause,
dest-clause, end-clause)
```

Where:

```
over-clause  ::= CPATH_OVER ',' pathptr ',' index ',' index ',' ...
dest-clause  ::= CPATH_DEST ',' pathptr ',' index ',' index ',' ...
tag-clause   ::= CPATH_TAG ',' tag
tag-clause   ::= CPATH_TAGS ',' tag ',' tag ',' ... ',' 0
tag-clause   ::= CPATH_TAGVEC ',' numtags ',' tagvector
data-clause  ::= CPATH_BYTES ',' vecsize ',' eltsize ',' data
red-clause   ::= CPATH_REDUCER ',' redfn
end-clause   ::= CPATH_END
```

The over-clause specifies the set of threads participating in the reduction. One or more of the indices should be CPATH_ALL, the wildcard value. All array elements matching the pattern are participating in the reduction. All participants must supply the same over-clause. The tags-clause specifies a vector of integer tags. All participants must supply the same tags. The reducer represents the function used to combine data pairwise. All participants must supply the same reducer. The data-clause specifies the input-data, which is an array of arbitrary-sized values. All participants must agree on the vecsize and eltsize. The dest-clause specifies the recipient of the reduced data (which may contain CPATH_ALL again). The data is sent to the recipient. The results can be received with CPathRecv using the same tags specified in the CPathReduce. The results may be analyzed with CPathMsgDecodeReduction, and freed with CPathMsgFree.

```
void CPathMsgDecodeReduction(void *msg, int *vecsize, int *eltsize, void
*bytes)
```

This function accepts an opaque CPath message which was created by a reduction. It locates the data within the message, and determines the vecsize and eltsize.

The function that combines elements pairwise must match this prototype, and be registered with the following registration function. It is the number returned by the registration function which must be passed to CPathReduce:

```
void ReduceFn(int vecsize, void *data1, void *data2)
```

The reduce function accepts two equally-sized arrays of input data. It combines the two arrays pairwise, storing the results in array 1.

```
int CPathRegisterReducer(void *fn)
```

Accepts a pointer to a reduction function, and returns an integer index for the function. This number can be used as a parameter to CPathReduce.

Contents

- *Projections*
 - *Generating Performance Traces*
 - * *Enabling Performance Tracing at Link/Run Time*
 - * *Controlling Tracing from Within the Program*
 - *The Projections Performance Visualization Tool*
 - * *Building Projections*
 - * *Visualization and Analysis using Projections*
 - * *Performance Views*
 - * *Miscellaneous features*
 - * *Known Issues*

15.1 Generating Performance Traces

Projections is a performance analysis/visualization framework that helps you understand and investigate performance-related problems in (Charm++) applications. It is a framework with an event tracing component which allows for control over the amount of information generated. The tracing has low perturbation on the application. It also has a Java-based visualization and analysis component with various views that help present the performance information in a visually useful manner.

Performance analysis with Projections typically involves two simple steps:

1. Prepare your application by linking with the appropriate trace generation modules and execute it to generate trace data.

2. Using the Java-based tool to visually study various aspects of the performance and locate the performance issues for that application execution.

The Charm++ runtime automatically records pertinent performance data for performance-related events during execution. These events include the start and end of entry method executions, message sends from entry methods and scheduler idle time. This means *most* users do not need to manually insert code into their applications in order to generate trace data. In scenarios where special performance information not captured by the runtime is required, an API (see section 15.1.2) is available for user-specific events with some support for visualization by the Java-based tool. If greater control over tracing activities (e.g. dynamically turning instrumentation on and off) is desired, the API also allows users to insert code into their applications for such purposes.

The automatic recording of events by the Projections framework introduces the overhead of an if-statement for each runtime event, even if no performance analysis traces are desired. Developers of Charm++ applications who consider such an overhead to be unacceptable (e.g. for a production application which requires the absolute best performance) may recompile the Charm++ runtime with the `--with-production` flag, which removes the instrumentation stubs. To enable the instrumentation stubs while retaining the other optimizations enabled by `--with-production`, one may compile Charm++ with both `--with-production` and `--enable-tracing`, which explicitly enables Projections tracing.

To enable performance tracing of your application, users simply need to link the appropriate trace data generation module(s) (also referred to as *tracemode(s)*). (see section 15.1.1)

15.1.1 Enabling Performance Tracing at Link/Run Time

Projections tracing modules dictate the type of performance data, detail, and format each processor will record. They are also referred to as “tracemodes.” There are currently 2 tracemodes available. Zero or more tracemodes may be specified at link-time. When no tracemodes are specified, no trace data is generated.

Tracemode projections

Link time option: `-tracemode projections`

This tracemode generates files that contain information about all Charm++ events like entry method calls and message packing during the execution of the program. The data will be used by Projections in visualization and analysis.

This tracemode creates a single symbol table file and *p* ASCII log files for *p* processors. The names of the log files will be `NAME.#.log{.gz}` where `NAME` is the name of your executable and `#` is the processor `#`. The name of the symbol table file is `NAME.sts` where `NAME` is the name of your executable.

This is the main source of data needed by the performance visualizer. Certain tools like timeline will not work without the detail data from this tracemode.

The following is a list of runtime options available under this tracemode:

- `+logsize NUM`: keep only `NUM` log entries in the memory of each processor. The logs are emptied and flushed to disk when filled. (defaults to 1,000,000)
- `+binary-trace`: generate projections log in binary form.
- `+gz-trace`: generate gzip (if available) compressed log files.
- `+no-gz-trace`: generate regular (uncompressed) log files.
- `+notracenested`: a debug option. Does not resume tracing outer entry methods when entry methods are nested (as can happen with `[local]` and `[inline]` calls).
- `+checknested`: a debug option. Checks if events are improperly nested while recorded and issue a warning immediately.

- `+trace-subdirs NUM`: divide the generated log files among NUM subdirectories of the trace root, each named `NAME.projdir.K`

Tracemode summary

Link time option: `-tracemode summary`

In this tracemode, execution time across all entry points for each processor is partitioned into a fixed number of equally sized time-interval bins. These bins are globally resized whenever they are all filled in order to accommodate longer execution times while keeping the amount of space used constant.

Additional data like the total number of calls made to each entry point is summarized within each processor.

This tracemode will generate a single symbol table file and p ASCII summary files for p processors. The names of the summary files will be `NAME.#.sum` where NAME is the name of your executable and # is the processor #. The name of the symbol table file is `NAME.sum.sts` where NAME is the name of your executable.

This tracemode can be used to control the amount of output generated in a run. It is typically used in scenarios where a quick look at the overall utilization graph of the application is desired to identify smaller regions of time for more detailed study. Attempting to generate the same graph using the detailed logs of the prior tracemode may be unnecessarily time consuming or resource intensive.

The following is a list of runtime options available under this tracemode:

- `+bincount NUM`: use NUM time-interval bins. The bins are resized and compacted when filled.
- `+binsize TIME`: sets the initial time quantum each bin represents.
- `+version`: set summary version to generate.
- `+sumDetail`: Generates a additional set of files, one per processor, that stores the time spent by each entry method associated with each time-bin. The names of “summary detail” files will be `NAME.#.sumd` where NAME is the name of your executable and # is the processor #.
- `+sumonly`: Generates a single file that stores a single utilization value per time-bin, averaged across all processors. This file bears the name `NAME.sum` where NAME is the name of your executable. This runtime option currently overrides the `+sumDetail` option.

General Runtime Options

The following is a list of runtime options available with the same semantics for all tracemodes:

- `+traceroot DIR`: place all generated files in DIR.
- `+traceoff`: trace generation is turned off when the application is started. The user is expected to insert code to turn tracing on at some point in the run.
- `+traceWarn`: By default, warning messages from the framework are not displayed. This option enables warning messages to be printed to screen. However, on large numbers of processors, they can overwhelm the terminal I/O system of the machine and result in unacceptable perturbation of the application.
- `+traceprocessors RANGE`: Only output logfiles for PEs present in the range (i.e. 0–31, 32–999966:1000, 999967–999999 to record every PE on the first 32, only every thousandth for the middle range, and the last 32 for a million processor run).

End-of-run Analysis for Data Reduction

As applications are scaled to thousands or hundreds of thousands of processors, the amount of data generated becomes extremely large and potentially unmanageable by the visualization tool. At the time of documentation, Projections

is capable of handling data from 8000+ processors but with somewhat severe tool responsiveness issues. We have developed an approach to mitigate this data size problem with options to trim-off “uninteresting” processors’ data by not writing such data at the end of an application’s execution.

This is currently done through heuristics to pick out interesting extremal (i.e. poorly behaved) processors and at the same time using a k -means clustering to pick out exemplar processors from equivalence classes to form a representative subset of processor data. The analyst is advised to also link in the summary module via `+tracemode summary` and enable the `+sumDetail` option in order to retain some profile data for processors whose data were dropped.

- `+extrema`: enables extremal processor identification analysis at the end of the application’s execution.
- `+numClusters`: determines the number of clusters (equivalence classes) to be used by the k -means clustering algorithm for determining exemplar processors. Analysts should take advantage of their knowledge of natural application decomposition to guess at a good value for this.

This feature is still being developed and refined as part of our research. It would be appreciated if users of this feature could contact the developers if you have input or suggestions.

15.1.2 Controlling Tracing from Within the Program

Selective Tracing

Charm++ allows users to start/stop tracing the execution at certain points in time on the local processor. Users are advised to make these calls on all processors and at well-defined points in the application.

Users may choose to have instrumentation turned off at first (by command line option `+traceoff` - see section 15.1.1) if some period of time in middle of the applications’ execution is of interest to the user.

Alternatively, users may start the application with instrumentation turned on (default) and turn off tracing for specific sections of the application.

Again, users are advised to be consistent as the `+traceoff` runtime option applies to all processors in the application.

- `void traceBegin()`
Enables the runtime to trace events (including all user events) on the local processor where `traceBegin` is called.
- `void traceEnd()`
Disables the runtime from tracing events (including all user events) on the local processor where `traceEnd` is called.

Explicit Flushing

By default, when linking with `-tracemode projections`, log files are flushed to disk whenever the number of entries on a processor reaches the logsize limit (see Section 15.1.1). However, this can occur at any time during the execution of the program, potentially causing performance perturbations. To address this, users can explicitly flush to disk using the `traceFlushLog()` function. Note that automatic flushing will still occur if the logsize limit is reached, but sufficiently frequent explicit flushes should prevent that from happening.

- `void traceFlushLog()`
Explicitly flushes the collected logs to disk.

User Events

Projections has the ability to visualize traceable user specified events. User events are usually displayed in the Timeline view as vertical bars above the entry methods. Alternatively the user event can be displayed as a vertical bar that vertically spans the timelines for all processors. Follow these following basic steps for creating user events in a Charm++ program:

1. Register an event with an identifying string and either specify or acquire a globally unique event identifier. All user events that are not registered will be displayed in white.
2. Use the event identifier to specify trace points in your code of interest to you.

The functions available are as follows:

- `int traceRegisterUserEvent(char* EventDesc, int EventNum=-1)`

This function registers a user event by associating `EventNum` to `EventDesc`. If `EventNum` is not specified, a globally unique event identifier is obtained from the runtime and returned. The string `EventDesc` must either be a constant string, or it can be a dynamically allocated string that is **NOT** freed by the program. If the `EventDesc` contains the substring "***" then the Projections Timeline tool will draw the event vertically spanning all PE timelines.

`EventNum` has to be the same on all processors. Therefore use one of the following methods to ensure the same value for any PEs generating the user events:

1. Call `traceRegisterUserEvent` on PE 0 in `main::main` without specifying an event number and store the returned event number into a readonly variable.
2. Call `traceRegisterUserEvent` and specify the event number on processor 0. Doing this on other processors has no effect. Afterwards, the event number can be used in the following user event calls.

Eg. `traceRegisterUserEvent("Time Step Begin", 10);`

Eg. `eventID = traceRegisterUserEvent("Time Step Begin");`

There are two main types of user events, bracketed and non bracketed. Non-bracketed user events mark a specific point in time. Bracketed user events span an arbitrary contiguous time range. Additionally, the user can supply a short user supplied text string that is recorded with the event in the log file. These strings should not contain newline characters, but they may contain simple html formatting tags such as `
`, ``, `<i>`, ``, etc.

The calls for recording user events are the following:

- `void traceUserEvent(int EventNum)`

This function creates a user event that marks a specific point in time.

Eg. `traceUserEvent(10);`

- `void traceBeginUserBracketEvent(int EventNum)`

`void traceEndUserBracketEvent(int EventNum)`

These functions record a user event spanning a time interval. The tracing framework automatically associates the call with the time it was made, so timestamps are not explicitly passed in as they are with `traceUserBracketEvent`.

- `void traceUserBracketEvent(int EventNum, double StartTime, double EndTime)`

This function records a user event spanning a time interval from `StartTime` to `EndTime`. Both `StartTime` and `EndTime` should be obtained from a call to `CmiWallTimer()` at the appropriate point in the program.

Eg.

```
traceRegisterUserEvent("Critical Code", 20); // on PE 0
double critStart = CmiWallTimer(); // start time
// do the critical code
traceUserBracketEvent(20, critStart, CmiWallTimer());
```

- `void traceUserSuppliedData(int data)`

This function records a user specified data value at the current time. This data value can be used to color entry method invocations in Timeline, see [15.2.3](#).

- `void traceUserSuppliedNote(char * note)`

This function records a user specified text string at the current time.

- `void traceUserSuppliedBracketedNote(char *note, int EventNum, double StartTime, double EndTime)`

This function records a user event spanning a time interval from `StartTime` to `EndTime`. Both `StartTime` and `EndTime` should be obtained from a call to `CmiWallTimer()` at the appropriate point in the program.

Additionally, a user supplied text string is recorded, and the `EventNum` is recorded. These events are therefore displayed with colors determined by the `EventNum`, just as those generated with `traceUserBracketEvent` are.

User Stats

Charm++ allows the user to track the progression of any variable or value throughout the program execution. These user specified stats can then be plotted in Projections, either over time or by processor. To enable this feature for Charm++, build Charm++ with the `-enable-tracing` flag.

Follow these steps to track user stats in a Charm++ program:

1. Register a stat with an identifying string and a globally unique integer identifier.
2. Update the value of the stat at points of interest in the code by calling the update stat functions.
3. Compile program with `-tracemode projections` flag.

The functions available are as follows:

- `int traceRegisterUserStat(const char * StatDesc, int StatNum)`

This function is called once near the beginning the of the Charm++ program. `StatDesc` is the identifying string and `StatNum` is the unique integer identifier.

- `void updateStat(int StatNum, double StatValue)`

This function updates the value of a user stat and can be called many times throughout program execution. `StatNum` is the integer identifier corresponding to the desired stat. `StatValue` is the updated value of the user stat.

- `void updateStatPair(int StatNum, double StatValue, double Time)`

This function works similar to `updateStat()`, but also allows the user to store a user specified time for the update. In Projections, the user can then choose which time scale to use: real time, user specified time, or ordered.

Function-level Tracing for Adaptive MPI Applications

Adaptive MPI (AMPI) is an implementation of the MPI interface on top of Charm++. As with standard MPI programs, the appropriate semantic context for performance analysis is captured through the observation of MPI calls

within C/C++/Fortran functions. Users can selectively begin and end tracing in AMPI programs using the routines `AMPI_Trace_begin` and `AMPI_Trace_end`.

15.2 The Projections Performance Visualization Tool

The Projections Java-based visualization tool (henceforth referred to as simply Projections) can be downloaded from the Charm++ website at <https://charm.cs.illinois.edu/software>. The directory which you download will henceforth be referred to as `PROJECTIONS_LOCATION`.

15.2.1 Building Projections

To rebuild Projections (optional) from the source:

1. Make sure your PATH contains the JDK commands “java”, “javac”, and “jar”, as well as the build tools “gradle” and “make”.
2. Make sure that you are using Java 8 or later. Do this by running “java -version” and “javac -version”.
3. From `PROJECTIONS_LOCATION/`, type “make”.
4. The following files are placed in ‘bin’:

```
projections : Starts projections, for UNIX machines
projections.bat : Starts projections, for Windows machines
projections.jar : archive of all the java and image files
```

15.2.2 Visualization and Analysis using Projections

Starting Up

From any location, type:

```
$ PROJECTIONS_LOCATION/bin/projections [NAME.sts]
```

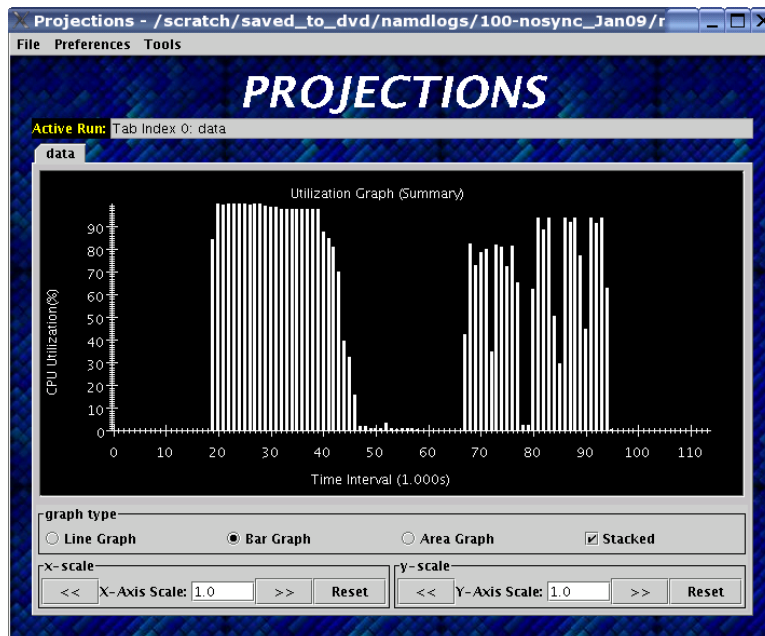
where `PROJECTIONS_LOCATION` is the path to the main projections directory.

Available options to the visualization component of Projections include:

- `-h` or `--help`: displays help information about available options.
- `-V` or `--version`: displays current Projections version number.
- `-u` or `--use-version <ver>`: overrides the data interpretation behavior of Projections to explicitly use *ver* instead of the current version. This is useful in scenarios where the latest version of Projections is not backward-compatible with older log formats.
- `-no-idle`: tells Projections to ignore idle time information contained in the logs.
- `-bgsz <x> <y> <z>`: tells Projections to compute additional derived information by assuming a logical 3-D Torus topology with the specified dimensionality and a processor-to-torus placement policy that matches Charm++’s placement policy on the BG/L class of machines. The presence of this option enables additional communication visualization options (see later). Note that this option is really meant to be used for logs generated from virtual-node mode BG/L executions. The additional information derived from any other logs would probably be misleading.

- `-print_usage`: tells Projections to also write to standard output the detailed graph numbers when viewing Usage Profiles (see later). This is useful for generating visuals that are better expressed by tools such as gnuplot than through screen captures of Projections plots.

Supplying the optional `NAME.sts` file in the command line will cause Projections to load data from the file at startup. This shortcut saves time selecting the desired dataset via the GUI's file dialog.



23: Projections main window

When Projections is started, it will display a main window as shown in figure 23. If summary (.sum) files are available in the set of data, a low-resolution utilization graph (Summary Display) will be displayed as shown. If summary files are not available, or if Projections was started without supplying the optional `NAME.sts` file, the main window will show a blank screen.

- **File** contains 3 options. *Open File(s)* allows you to explicitly load a data set. This happens if you had not specified a `NAME.sts` file in the command line when starting Projections or if you wish to explicitly load a new dataset. It brings up a dialog box that allows you to select the location of the dataset you wish to study. Navigate to the directory containing your data and select the .sts file. Click on “Open”. If you have selected a valid file, Projections will load in some preliminary data from the files and then activate the rest of the options under the menu item **Tools**. *Close current data* currently works the same way as *Close all data*. They unload all current Projections data so one can load in a new set of data. They will also deactivate the individual items provided in the **Tools** menu option.
- **Preferences** generally allows you to set foreground or background colors and entry method color schemes. This is useful for configuring the color schemes of Projections windows to be print-friendly.
- **Tools** lists the set of available tools for analysis of generated trace data. It will be described in great detail under section 15.2.2.

The Summary Display loaded on the Main Window displays basic processor utilization data (averaged across all processors) over time intervals. This is provided by the data generated by the summary tracemode. This view offers no special features over and above the **Standard Graph Display** described in section 15.2.4. Please refer the appropriate section on information for using its available features.

There should not be any serious performance issues involved in the loading of summary data on the main window.

Available Tools

The following tools and views become available to you after a dataset has been loaded (with the exception of Multirun Analysis) and may be accessed via the menu item Tools:

- The **Graphs** view is where you can analyze your data by breaking it into any number of intervals and look at what goes on in each of those intervals.
- The **Timelines** view lets you look at what a specific processor is doing at each moment of the program. It is the most detailed view of a parallel application Projections offers (and correspondingly, the most resource-hungry).
- The **Usage Profile** view lets you see percentage-wise what entry methods each processor spends its time on during a specified time range. It is particularly useful for identifying load imbalance and the probable offending entry method.
- The **Communication** view is a general tool that presents communication properties contributed by each entry point across the processors.
- The **Log File Viewer** provides a human-readable, verbose interpretation of a log file's entries.
- The **Histograms** view presents entry point or communication histogram information (ie. the frequency of occurrence of events given an activity property like time bin size or message size on the x-axis).
- The **Overview** view gives user an overview of the utilization of all processors during the execution. It is an extremely useful initial tool to begin your performance analysis efforts with as it provides an overall picture of application performance while being very light-weight at the same time.
- The **Animation** view animates the processor usage over a specified range of time and a specified interval size.
- The **Time Profile Graph** view is a more detailed presentation of the **Graphs** utilization view in that it presents the time contribution by each entry point across the desired time interval. While the **Graphs** view can show the same data, it is unable to stack the entry points, which proves useful in some cases.

15.2.3 Performance Views

Graphs

The Graphs window (see figure 24) is where you can analyze your data by breaking it into any number of intervals and look at what goes on in each of those intervals.

When the Graph Window first appears, a dialog box will also appear. It will ask for the following information (Please refer to [Miscellaneous features](#) for information on special features you can use involving the various fields):

- Processor(s): Choose which processor(s) you wish to visualize graph information for.
- Start Time : Choose the starting time of interest. A time-based field.
- End Time : Choose the ending time of interest. A time-based field.
- Interval Size : Determine the size of an interval. The number of intervals will also be determined by this value (End Time - Start Time divided by Interval Size). A time-based field.

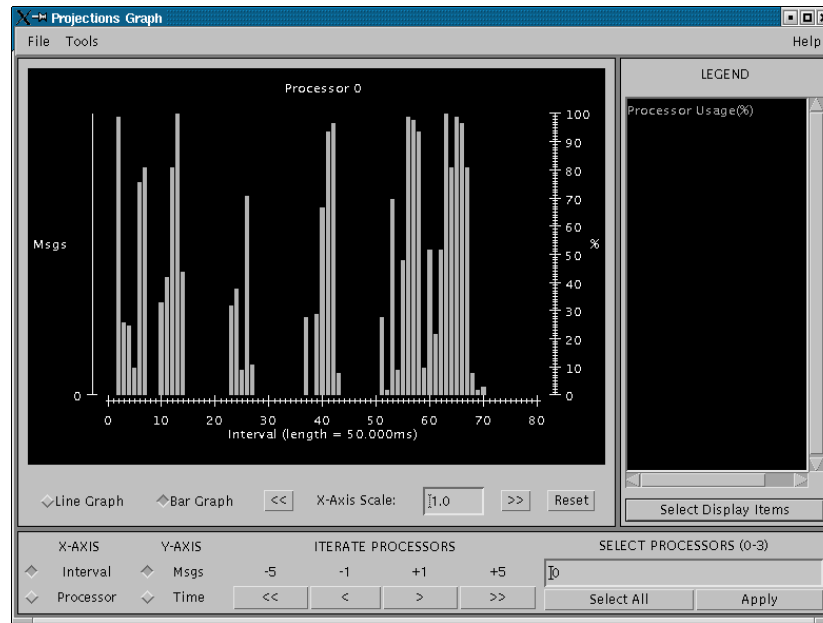
Standard Projections dialog options and buttons are also available (see [Miscellaneous features](#) for details).

The following menu items are available:

- **File** contains 2 options: *Print Graph* uses Java's built-in print manager feature to render the tool's displays (usually to a printer or a file depending on the platform on which Java is supported). Note that the current implementation of the renderer does not behave in exactly the same fashion as the screen renderer, so you should expect the output to look somewhat different. *Close* simply closes the Graph window.

- **Tools** contains 2 options: *Set Interval Size* reloads the dialog box so you could select a new time range over which to view Graph data. *Timeline* is currently not implemented. Its intended as a convenient way to load Timeline data (see section 15.2.3) over the same parameters as the current Graph view.

The amount of time to analyze your data depends on several factors, including the number of processors, number of entries, and number of intervals you have selected. A progress meter will show the amount of data loaded so far. The meter will not, however, report rendering progress which is determined mainly by the number of intervals selected. As a rule of thumb, limit the number of intervals to 1,000 or less.



24: Graph tool

The Graph Window has 3 components in its display:

1. **Display Panel** (located : top-left area)

- Displays title, graph, and axes. To the left is a y-axis bar for detailed information involving the number of messages sent or time executed depending on the **Control Panel** toggle selected (see below). To the right is a y-axis bar for average processor-utilization information. The x-axis may be based on time-interval or per-processor information depending on the appropriate **Control Panel** toggle.
- Allows you to toggle display between a line graph and a bar graph.
- Allows you to scale the graph along the X-axis. You can either enter a scale value ≥ 1.0 in the text box, or you can use the << and >> buttons to increment/decrement the scale by .25. Clicking on Reset sets the scale back to 1.0. When the scale is greater than 1.0, a scrollbar will appear along the bottom of the graph to let you scroll back and forth.

2. **Legend Panel** (located : top-right area)

- Shows what information is currently being displayed on the graph and what color represents that information.
- Click on the 'Select Display Items' button to bring up a window to add/remove items from the graph and to change the colors of the items:
 - The **Select Display Items** window shows a list of items that you can display on the graph. There are 3 main sections: System Usage, System Msgs, and User Entries. The System Usage and System Msgs are the same for all programs. The User Entries section has program-specific items in it.

- Click on the checkbox next to an item to have it displayed on the graph.
- Click on the colorbox next to an item to modify its color.
- Click on ‘Select All’ to choose all of the items
- Click on ‘Clear All’ to remove all of the items
- Click on ‘Apply’ to apply you choices/changes to the graph
- Click on ‘Close’ to exit

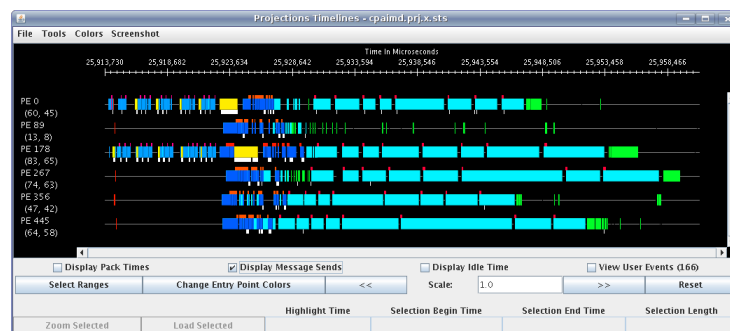
3. Control Panel (located : bottom area)

- Allows you to toggle what is displayed on the X-axis. You can either have the x-axis display the data by interval or by processor.
- Allows you to toggle what is displayed on the Y-axis. You can either have the y-axis display the data by the number of msgs sent or by the amount of time taken.
- Allows you to change what data is being displayed by iterating through the selections. If you have selected an x-axis type of ‘interval’, that means you are looking at what goes on in each interval for a specific processor. Clicking on the <<, <, >, >> buttons will change the processor you are looking at by either -5, -1, +1, or +5. Conversely, if you have an x-axis of ‘processor’, then the iterate buttons will change the value of the interval that you are looking at for each processor.
- Allows you to indicate which intervals/processors you want to examine. Instead of just looking at one processor or one interval, the box and buttons on the right side of this panel let you choose any number or processors/intervals to look at. This field behaves like a processor field. Please refer to section 15.2.4 for more information about the special features on using processor fields.

Clicking on ‘Apply’ updates the graph with your choices. Clicking on ‘Select All’ chooses the entire processor range. When you select more than one processor’s worth of data to display, the graph will show the desired information summed across all selected processors. The exception to this is processor utilization data which is always displayed as data averaged across all selected processors.

Timelines

The Timeline window (see figure 25) lets you look at what a specific processor is doing at each moment of the program.



25: Timeline Tool

When opening a Timeline view, a dialog box appears. The box asks for the following information (Please refer to 15.2.4 for information on special features you can use involving the various fields):

- Processor(s): Choose which processor(s) you want to see in the timeline.
- Start Time : Choose what time you want your timeline to start at. A time-based field.

- End Time : Choose what time you want your timeline to end at. A time-based field.

Standard Projections dialog options and buttons are also available (see [15.2.4](#) for details).

The following menu options are available:

- **File** contains 1 enabled option: *Close* simply closes the Timeline Window.
- **Tools** contains 1 option: *Modify Ranges* opens the initial dialog box thereby allowing you to select new set of processors or time duration parameters.
- **Colors** contains options for loading, using, and modifying color schemes. *Change Colors* functions in a manner similar to the button of the same name described under control panel information below. *Save Colors* allows you to save the current color set to a file named “color.map” into the same directory where your data logs are stored. Note that the directory must have write permissions for you before this can work. We are currently working on a more flexible scheme for storing saved color sets. *Restore Colors* allows you to load any previously saved color sets described above. *Default Colors* resets the current color set to the default set that Projections assigns without user intervention.

Other color schemes are provided that can be used for some applications. The colors set as described above are the default coloring scheme. Other options for coloring the events are by event ID (chare array index), user supplied parameter, or memory usage. In order to color by a user supplied parameter such as timestep, the C function `traceUserSuppliedData(int value);` should be called within some entry methods. If such a method is called in an entry method, the entry method invocation can be colored by the parameter. The user supplied data can also be viewed in the tooltip that appears when the cursor hovers over an entry method invocation in the window. To color by memory usage, the C function `traceMemoryUsage();` should be called in all entry methods. The call records the current memory usage. Red indicates high memory usage, and green indicates low memory usage. The actual memory usage can also be viewed in the tooltips that appear when the cursor is over an event. The memory usage is only available in when using a Charm++ version that uses GNU memory.

- **Screenshot** contains 1 option: *Save as JPG or PNG* save the visible portion of the visualization to an image file. You must choose a filename ending with a `.png` or `.jpg` extension when choosing the location to save the image. The appropriate filetype is chosen based on the chosen filename extension. If the view is zoomed in, only the portion currently shown on screen is saved.

The Timeline Window consists of two parts:

1. **Display Panel** (located: top area)

This is where the timelines are displayed and is the largest portion of the window. The time axis is displayed at the top of the panel. The left side of the panel shows the processor labels, each containing a processor number and two strange numbers. These two numbers represent the percentage of the loaded timeline during which work occurs. The first of the two numbers is the “non-idle” time, i.e. the portion of the time in the timeline not spent in idle regions. This contains both time for entry methods as well as other uninstrumented time spent likely in the Charm++ runtime. The second number is the percentage of the time used by the entry methods for the selected range.

The timeline itself consists of colored bars for each event. Placing the cursor over any of these bars will display information about the event including: the name, the begin time, the end time, the total time, the time spent packing, the number of messages it created, and which processor created the event.

Hovering over an event bar will cause a window to popup. This window contains detailed information about the messages sent by the clicked upon event.

Clicking on an event bar will cause a line to be drawn to the beginning of the event bar from the point where the message causing the event originated. This option may not be applicable for threaded events. If the message originated on a processor not currently included in the visualization, the other processor will be loaded, and then the message line will be drawn. A warning message will appear if the message origination point is outside the time duration, and hence no line will be drawn.

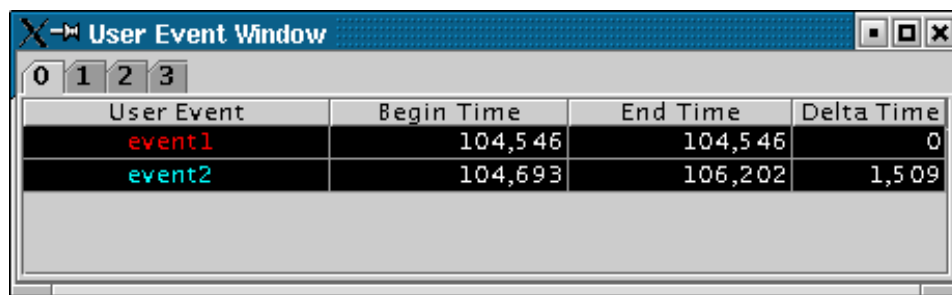
User events are displayed as bars above the ordinary event bars in the display area. If the name of the user event contains a substring “***” then the bar will vertically span the whole screen.

Message pack times and send points can be displayed below the event bars. The message sends are small white tick marks, while the message pack times are small pink bars usually occurring immediately after the message send point. If zoomed in to a point where each microsecond takes more than one pixel, the message send point and the following packing time may appear disconnected. This is an inherent problem with the granularity used for the logfiles.

2. Control Panel (located: bottom area)

The controls in this panel are obvious, but we mention one here anyway.

View User Event - Checking this box will bring up a new window showing the string description, begin time, end time and duration of all user events on each processor. You can access information on user events on different processors by accessing the numbered tabs near the top of the display.



The screenshot shows a window titled "User Event Window" with a tabbed interface. The first tab is selected, showing a table with the following data:

User Event	Begin Time	End Time	Delta Time
event1	104,546	104,546	0
event2	104,693	106,202	1,509

26: User Event Window

Various features appear when the user moves the mouse cursor over the top axis. A vertical line will appear to highlight a specific time. The exact time will be displayed at the bottom of the window. Additionally a user can select a range by clicking while a time is highlighted and dragging to the left or right of that point. As a selection is being made, a vertical white line will mark the beginning and end of the range. Between these lines, the background color for the display will change to gray to better distinguish the selection from the surrounding areas. After a selection is made, its duration is displayed at the bottom. A user can zoom into the selection by clicking the “Zoom Selected” button. To release a selection, single-click anywhere along the axis. Clicking “Load Selected” when a selection is active will cause the timeline range to be reloaded. To zoom out, the “<<” or “Reset” button can be used.

To then zoom into the selected area via this interface, click on either the “Zoom Selected” or the “Load Selected” buttons. The difference between these two buttons is that the “Load Selected” zooms into the selected area and discards any events that are outside the time range. This is more efficient than “Zoom Selected” as the latter draws all the events on a virtual canvas and then zooms into the canvas. The disadvantage of using “Load Selected” is that it becomes impossible to zoom back out without having to re-specify the time range via the “Select Ranges” button.

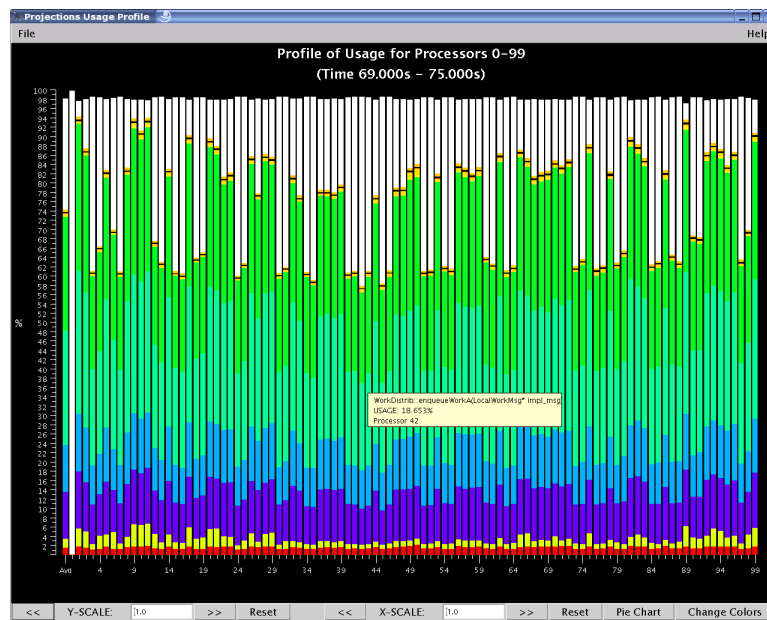
Performance-wise, this is the most memory-intensive part of the visualization tool. The load and zoom times are proportional to the number of events displayed. The user should be aware of how event-intensive the application is over the desired time-period before proceeding to use this view. If Projections takes too long to load a timeline, cancel the load and choose a smaller time range or fewer processors. We expect to add features to alleviate this problem in future releases.

Usage Profile

The Usage Profile window (see figure 27) lets you see percentage-wise what each processor spends its time on during a specified period.

When the window first comes up, a dialog box appears asking for the processor(s) you want to look at as well as the time range you want to look at. This dialog functions in exactly the same way as for the Timeline tool (see section

15.2.3).



27: Usage Profile

The following menu options are available in this view:

- **File** has 2 options: *Select Processors* reloads the dialog box for the view and allows you to select a new processor and time range for this view. *Print Profile* currently does nothing. This will be addressed in a later release of Projections.

The following components are supported in this view:

- **Main Display** (located: top area) The left axis of the display shows a scale from 0% to 100%. The main part of the display shows the statistics. Each processor is represented by a vertical bar with the leftmost bar representing the statistics averaged across all processors. The bottom of the bar always shows the time spent in each entry method (distinguished by the entry method's assigned color). Above that is always reported the message pack time (in black), message unpack time (in orange) and idle time (in white). Above this, if the information exists, are colored bars representing communication CPU overheads contributed by each entry method (again, distinguished by the same set of colors representing entry methods). Finally the black area on top represents time overheads that the Charm++ runtime cannot account for.

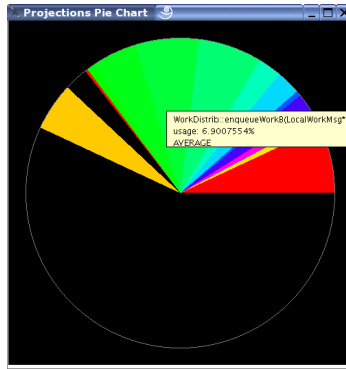
If you mouse-over a portion of the bar (with the exception of the black area on top), a pop-up window will appear telling you the name of the item, what percent of the usage it has, and the processor it is on.

- **Control Panel** (located: bottom area) Allows for adjustments of the scales in both the X and Y directions. The X direction is useful if you are looking at a large number of processors. The Y direction is useful if there are small-percentage items for a processor. The “Reset” button allows you to reset the X and Y scales.

The “Pie Chart” button generates a pie chart representation (see figure 28) of the same information using averaged statistics but without idle time and communication CPU overheads.

The “Change Colors” button lists all entry methods displayed on the main display and their assigned colors. It allows you to change those assigned colors to aid in highlighting entry methods.

The resource consumption of this view is moderate. Load times and visualization times should be relatively fast, but dismissing the tool may result in a very slight delay while Projections reclaims memory through Java's garbage collection system.



28: Pie Chart representation of average usage

Communication

The communication tool (see figure 29) visualizes communication properties on each processor over a user-specified time range.

The dialog box of the tool allows you to specify the time period within which to load communication characteristics information. This dialog box is exactly the same as that of the Timeline tool (see section 15.2.3).

The main component employs the standard capabilities provided by Projections' standard graph (see 15.2.4).

The control panel allows you to switch between the following communication characteristics:

- Number of messages sent by entry methods (initial default view)
- Number of bytes sent by entry methods
- Number of messages received by entry methods
- Number of bytes received by entry methods
- Number of messages sent externally (physically) by entry methods
- Number of bytes sent externally (physically) by entry methods
- Number of hops messages traveled before being received by an entry methods. This is available when the runtime option `-bgsiz` (See section 15.2.2) is supplied.

This view uses memory proportional to the number of processors selected.

Communication vs Time

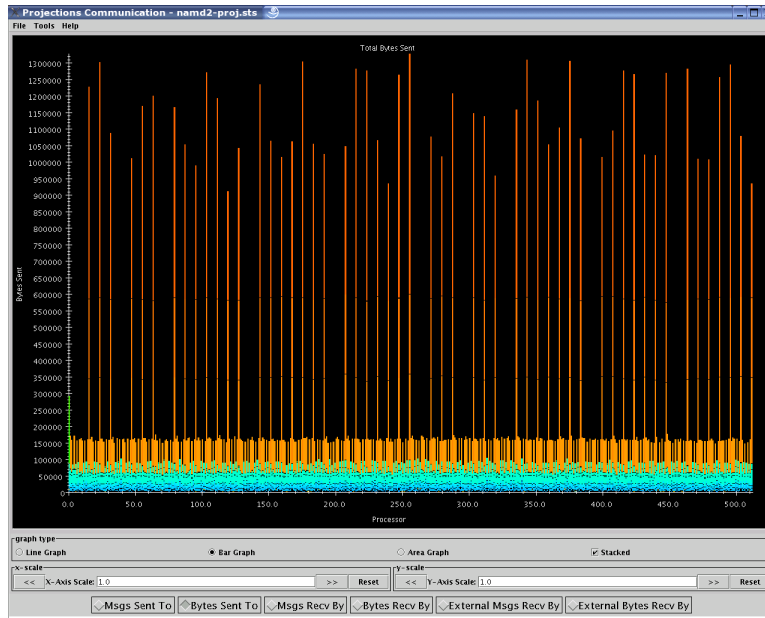
The communication over time tool (see figure 30) visualizes communication properties over all processors and displayed over a user-specified time range on the x-axis.

The dialog box of the tool allows you to specify the time period within which to load communication characteristics information. This dialog box is exactly the same as that of the Communication tool (see section 15.2.3).

The main component employs the standard capabilities provided by Projections' standard graph (see 15.2.4).

The control panel allows you to switch between the following communication characteristics:

- Number of messages sent by entry methods (initial default view)
- Number of bytes sent by entry methods
- Number of messages received by entry methods



29: Communication View

- Number of bytes received by entry methods
- Number of messages sent externally (physically) by entry methods
- Number of bytes sent externally (physically) by entry methods
- Number of hops messages travelled before being received by an entry methods (available only on trace logs generated on the Bluegene machine).

This view has no known problems loading any range or volume of data.

View Log Files

This window (see figure 31) lets you see a translation of a log file from a bunch of numbers to a verbose version. A dialog box asks which processor you want to look at. After choosing and pressing OK, the translated version appears. Note that this is *not* a standard processor field. This tool will only load *exactly* one processor's data.

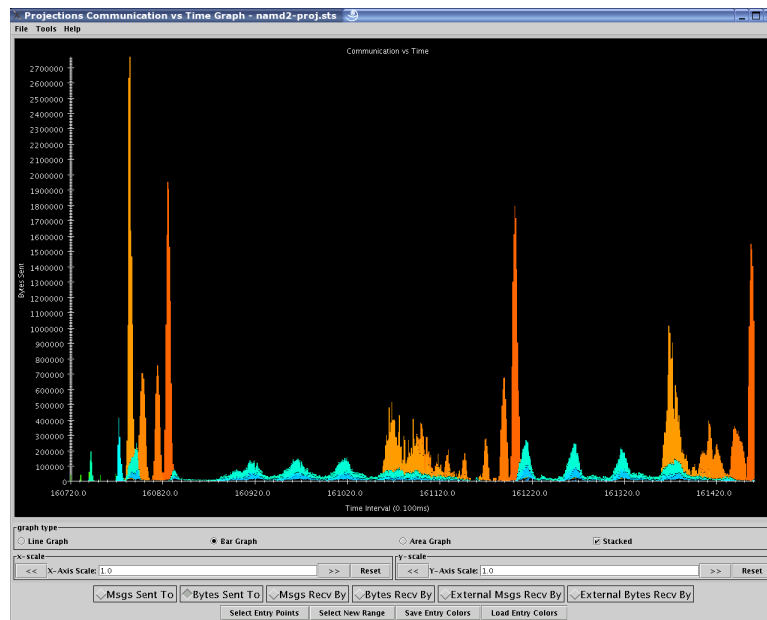
Each line has:

- a line number (starting at 0)
- the time the event occurred at
- a description of what happened.

This tool has the following menu options:

- **File** has 2 options: *Open File* reloads the dialog box and allows the user to select a new processor's data to be loaded. *Close* closes the current window.
- **Help** has 2 options: *Index* currently does not do anything. This will be addressed in a later release of Projections. *About* currently does not do anything. This will also be addressed in a later release of Projections.

The tool has 2 buttons. "Open File" reloads the dialog box (described above) and allows the user to select a new processor's data to be loaded. "Close Window" closes the current window.



30: Communication View over Time

The figure shows a window titled "Projections Log File Viewer". The main area displays a log file for processor 0. The log file has a header "LOG FILE FOR PROCESSOR 0" and a table with columns "LINE", "TIME", and "EVENT". The log file contains 15 entries, each with a line number, a time value, and an event description. At the bottom of the window, there are two buttons: "Open File" and "Close Window".

LINE	TIME	EVENT
0	5735699	END PROCESSING of message sent to dummy_thread_chare
1	5735703	IDLE begin
2	5735727	IDLE end
3	5735730	IDLE begin
4	5736055	IDLE end
5	5736455	IDLE begin
6	5736464	IDLE end
7	5736467	IDLE begin
8	5737309	IDLE end
9	5737313	IDLE begin
10	5737336	IDLE end
11	5737339	IDLE begin
12	5737401	IDLE end
13	5737405	IDLE begin
14	5737428	IDLE end

31: Log File View

Histograms

This module (see figure 32) allows you to examine the performance property distribution of all your entry points (EP). It gives a histogram of different number of EPs that have the following properties falling in different property bins:

The dialog box for this view asks the following information from the user. (Please refer to 15.2.4 for information on special features you can use involving the various fields):

- **Processor(s):** Choose which processor(s) you wish to visualize histogram information for.
- **Start Time:** Choose the starting time of interest. A time-based field.
- **End Time:** Choose the ending time of interest. A time-based field.
- **Number of Bins:** Select the number of property bins to fit frequency data under. A simple numeric field.
- **Size of Bin:** Determine (in units - microseconds or bytes) how large each bin should be. A simple numeric field.
- **Starting Bin Size:** Determine (in units - microseconds or bytes) how far to offset the data. This is useful for ignoring data that is too small to be considered significant, but could overwhelm other data because of the sheer numbers of occurrences. A simple numeric field.

The dialog box reports the selection of bins as specified by the user by displaying the minimum bin size (in units - microseconds or bytes) to the maximum bin size. “units” refer to microseconds for time-based histograms or bytes for histograms representing message sizes.

Standard graph features can be employed for the main display of this view (see section 15.2.4).

The following menu items are available in this tool:

- **File** offers 3 options: *Select Entry Points* currently does nothing. It is intended to behave similarly to the button “Select Entries” described below. This will be fixed in a later release of Projections. *Set Range* reloads the dialog box and allows the user to load data based on new parameters. *Close* closes the current tool window.
- **View** provides 1 option: *Show Longest EPs* currently does nothing. It is intended to behave similarly to the button “Out-of-Range EPs” and will be fixed in a later release of Projections.

The following options are available in the control panel in the form of toggle buttons:

- Entry method execution time (How long did that entry method ran for?)
- Entry method creation message size (How large was the message that caused the entry method’s execution?)

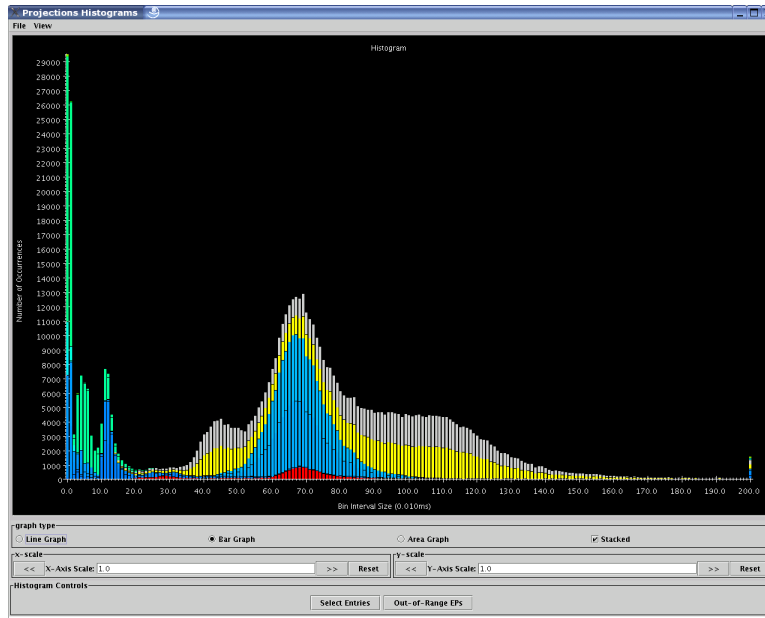
The use of the tool is somewhat counterintuitive. The dialog box is created immediately and when the tool window is created, it is defaulted to a time-based histogram. You may change this histogram to a message-size-based histogram by selecting the “Message Size” radio button which would then update the graph using the same parameters provided in the dialog box. This issue will be fixed in upcoming editions of Projections.

The following features are, as of this writing, not implemented. They will be ready in a later release of Projections.

The “Select Entries” button is intended to bring up a color selection and filtering window that allows you to filter away entry methods from the count. This offers more control over the analysis (e.g. when you already know EP 5 takes 20-30ms and you want to know if there are other entry points also takes 20-30ms).

The “Out-of-Range EPs” button is intended to bring up a table detailing all the entry methods that fall into the overflow (last) bin. This list will, by default, be listed in descending order of time taken by the entry methods.

The performance of this view is affected by the number of bins the user wishes to analyze. We recommend the user limits the analysis to 1,000 bins or less.



32: Histogram view

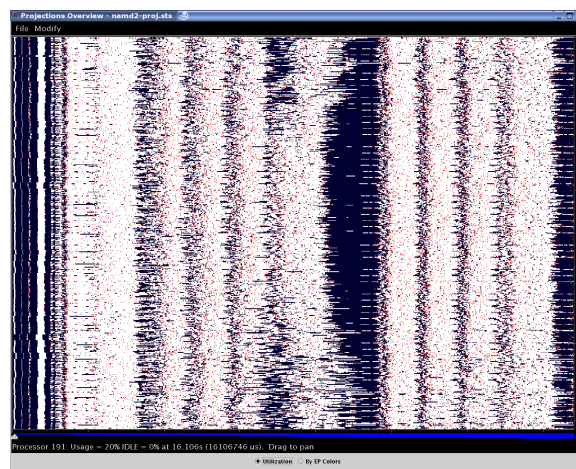
Overview

Overview (see figure 33) gives users an overview of the utilization of all processors during the execution over a user-specified time range.

The dialog box of the tool allows you to specify the time period within which to load overview information. This dialog box is exactly the same as that of the Timeline tool (see section 15.2.3).

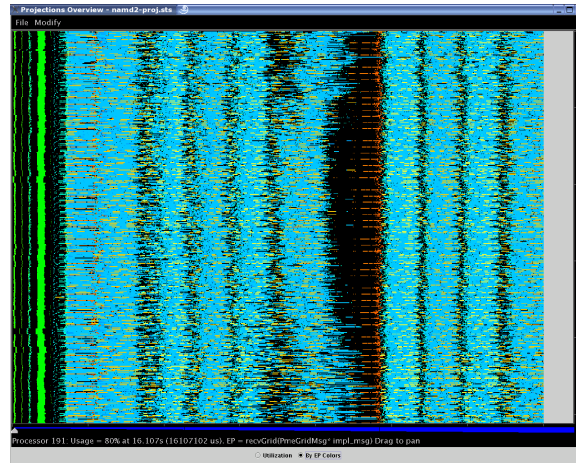
This tool provides support for the following menu options:

- **File** provides 1 option: *Close* closes the current tool.
- **Modify** provides 1 option: *Set Range* reloads the dialog box and allows the user to specify new parameters for rendering new overview information.



33: Different Overview presentation forms (1) - Overview.

The view currently hard codes the number of intervals to 7,000 independent of the time-range desired.



34: Different Overview presentation forms (2) - Overview with dominant Entry Method colors.

Each processor has a row of colored bars in the display, different colors indicating different utilization at that time (white representing 100 utilization and black representing idle time). The usage of a processor at the specific time is displayed in the status bar below the graph. Vertical and horizontal zoom is enabled by two zooming bars to the right and lower of the graph. Panning is possible by clicking on any part of the display and dragging the mouse.

The “by EP colors” radio button provides more detail by replacing the utilization colors with the colors of the most significant entry method execution time in that time-interval on that processor represented by the cells (as illustrated in figure 34).

The Overview tool uses memory proportional to the number of processors selected. If an out-of-memory error is encountered, try again by skipping processors (e.g. 0-8191:2 instead of 0-8191). This should show the general application structure almost as well as using the full processor range.

Animations

This window (see figure 35) animates the processor usage over a specified range of time and a specified interval size.

The dialog box to load animation information is exactly the same as that of the Graph tool (see section 15.2.3).

A color temperature bar serves as a legend for displaying different processor utilization as the animation progresses. Each time interval will have its data rendered as a frame. A frame displays in text on the top of the display the currently represented execution time of the application and what the size of an interval is.

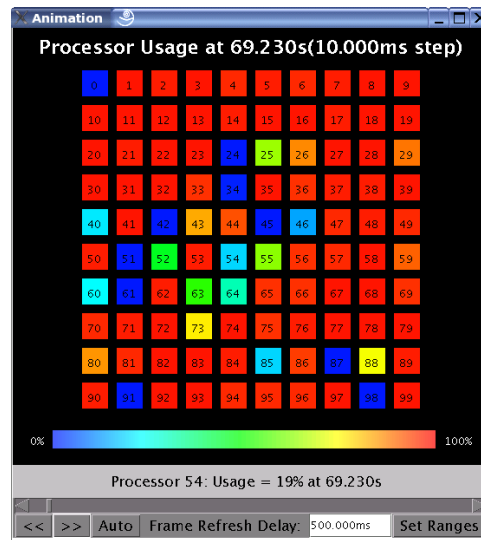
Each selected processor is laid out in a 2-D plot as close to a square as possible. The view employs a color temperature ranging from blue (cool - low utilization) to bright red (hot - high utilization) to represent utilization.

You may manually update the frames by using the “<<” or “>>” buttons to visualize the preceding or next frames respectively. The “Auto” button toggles automatic animation given the desired refresh rate.

The “Frame Refresh Delay” field allows you to select the real time delay between frames. It is a time-based field (see section 15.2.4 for special features in using time-based fields).

The “Set Ranges” button allows you to set new parameters for this view via the dialog box.

This view has no known performance issues.

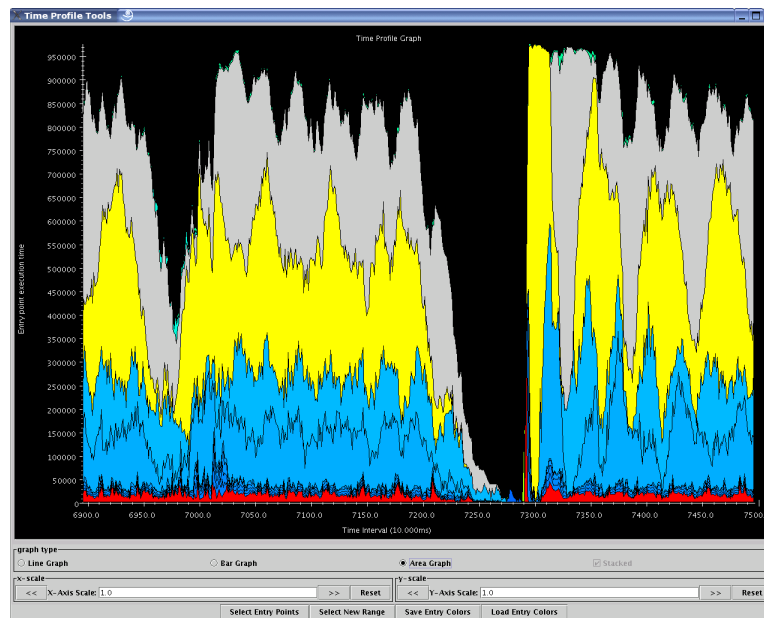


35: Animation View

Time Profile Graph

The Time Profile view (see figure 36) is a visualization of the amount of time contributed by each entry method summed across all processors and displayed by user-adjustable time intervals.

Time Profile's dialog box is exactly the same as that of the Graph tool (see section 15.2.3).



36: Time Profile Graph View

Standard graph features can be employed for the main display of this view (see section 15.2.4).

Under the tool options, one may:

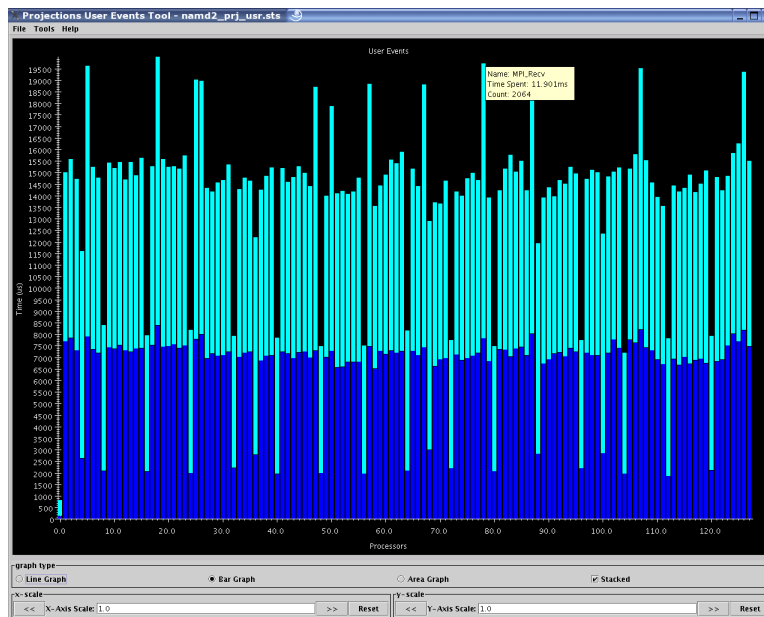
- Filter the set of entry methods to be displayed on the graph via the “Select Entry Points” button. One may also modify the color set used for the entry methods via this option.

- Use the “Select New Range” button to reload the dialog box for the tool and set new parameters for visualization (eg. different time range, different set of processors or different interval sizes).
- Store the current set of entry method colors to disk (to the same directory where the trace logs are stored). This is done via the “Save Entry Colors” button.
- Load the stored set of entry method colors (if it exists) from disk (from the same directory where the trace logs are stored). This is done via the “Load Entry Colors” button.

This tool’s performance is tied to the number of intervals desired by the user. We recommend that the user stick to visualizing 1,000 intervals or less.

User Events Profile

The User Events view is essentially a usage profile (See section 15.2.3) of bracketed user events (if any) that were recorded over a specified time range. The x-axis holds bars of data associated with each processor while the y-axis represents the time spent by each user event. Each user event is assigned a color.

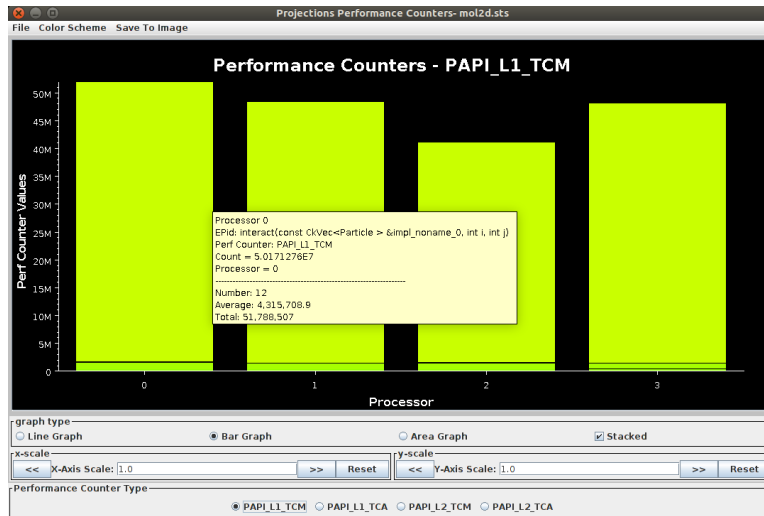


37: User Events Profile View

It is important to note that user-events can be arbitrarily nested. The view currently displays information based on raw data without regard to the way the events are nested. Memory usage is proportional to the number of processors to be displayed.

Performance Counters

This view is enabled when Charm++ is compiled with the `papi` option, which enables the collection of performance counter data using PAPI. Currently, this collects the highest level cache misses and accesses available on the system (PAPI_L1_TCM and PAPI_L1_TCA, PAPI_L2_TCM and PAPI_L2_TCA, or PAPI_L3_TCM and PAPI_L3_TCA) on all platforms except Cray, where it collects PAPI_FP_OPS, PAPI_TOT_INS, `perf::PERF_COUNT_HW_CACHE_LL:MISS`, `DATA_PREFETCHER:ALL`, PAPI_L1_DCA and PAPI_TOT_CYC. This tool shows the values of the collected performance counters on different PEs, broken down by entry point. An example is shown in Figure 38.



38: Performance Counters View

Outlier Analysis

For performance logs generated from large numbers of processors, it is often difficult to view in detail the behavior of poorly behaved processors. This view attempts to present information similar to usage profile but only for processors whose behavior is “extreme”.

Select Range

Valid Processors = 0-63
Processors : 0-63

Valid Time Range = 0 to 31.268s
Start Time : 0 End Time : 31.268s
Total Time selected : 31.268s

Attribute: Execution Time by Activity
Activity: PROJECTIONS

Outlier Threshold: 7 Processors
Number of Clusters: 5

Save History to Disk
Add to History List Remove selected History
OK Update Cancel

39: Outlier Analysis Selection Dialog

“Extreme” processors are identified through the application of heuristics specific to the attribute that analysts wish to study applied to a specific activity type. You can specify the number of “extreme” processors are to be picked out by Projections by filling the appropriate number in the field “Outlier Threshold”. The default is to pick 10% of the total number of processors up to a cap of 20. As an example, an analyst may wish to find “extreme” processors with respect to the idle time of normal Charm++ trace events.

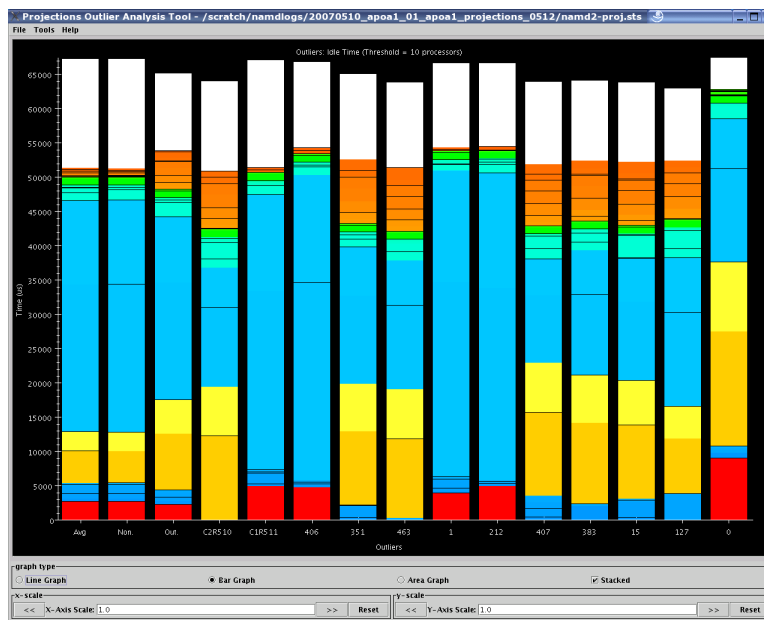
Figure 39 shows the choices available to this tool. Specific to this view are two pull-down menus: *Attribute* and *Activity*.

There are two *Activity* options:

1. The *Projections* activity type refer to the entry methods executed by the Charm++ runtime system.
2. The *User Events* activity type refer to records of events as captured through `traceUserEvent`-type calls described in section 15.1.2.

There are four *Attribute* options:

1. *Execution time by Activity* tells the tool to apply heuristics based on the execution time of each instance of an activity occurring within the specified time range.
2. *Idle time* tells the tool to apply a simple sort over all processors on the least total idle time recorded. This will work only for the *Projections* activity type.
3. *Msgs sent by Activity* tells the tool to apply heuristics based on the number of messages sent over each instance of an activity occurring within the specified time range. This option is currently not implemented but is expected to work over all activity types.
4. *Bytes sent by Activity* tells the tool to apply heuristics based on the size (in bytes) of messages sent over each instance of an activity occurring within the specified time range. This option is currently not implemented but is expected to work over all activity types.



40: Outlier Analysis View

At the same time, a k -means clustering algorithm is applied to the data to help identify processors with exemplar behavior that is representative of each cluster (or equivalence class) identified by the algorithm. You can control the value of k by filling in the appropriate number in the field “Number of Clusters”. The default value is 5.

The result of applying the required heuristics to the appropriate *attribute* and *activity* types results in a chart similar to figure 40. This is essentially a usage profile that shows, over the user’s selected time range, from left to right:

- A bar representing the global average of execution time of each activity over all processors.
- A bar representing the average activity profile of all non-outlier (or non-extreme) processors.
- A bar representing the average activity profile of all outlier (or extreme) processors identified by the heuristics.

- Bars representing the activity profile of a representative processor from each cluster of processors identified by the application of the k -means clustering algorithm.
- Bars representing the activity profile of each identified outlier processor, sorted in order of significance (right-most processor bar is the most significant).

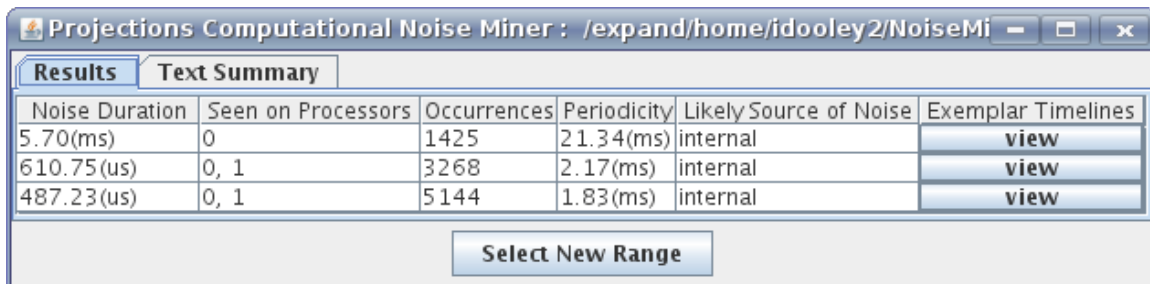
The tool helps the user reduce the number of processor bars that must be visually examined in order to identify candidates for more detailed study. To further the cause of this goal, if the analyst has the *timeline* view (see section 15.2.3) open, a mouse-click on any of the processor activity profile bars (except for group-averaged bars) will load that processor's detailed timeline (over the time range specified in the timeline view) into the timeline view itself.

Online Live Analysis

Projections provides a continuous performance monitoring tool - CCS streaming. Different from other tools discussed above, which are used to visualize post-mortem data, CCS streaming visualizes the running programs. In order to use it, the Charm++ program needs to be linked with `-tracemode utilization`. The command line needs to include `++server ++server-port <port>`. `<port>` is the socket port number on the server side.

Multirun Analysis

Noise Miner



The screenshot shows a window titled "Projections Computational Noise Miner : /expand/home/idooley2/NoiseMi". It has two tabs: "Results" (selected) and "Text Summary". The "Results" tab contains a table with the following data:

Noise Duration	Seen on Processors	Occurrences	Periodicity	Likely Source of Noise	Exemplar Timelines
5.70(ms)	0	1425	21.34(ms)	internal	view
610.75(us)	0, 1	3268	2.17(ms)	internal	view
487.23(us)	0, 1	5144	1.83(ms)	internal	view

Below the table is a button labeled "Select New Range".

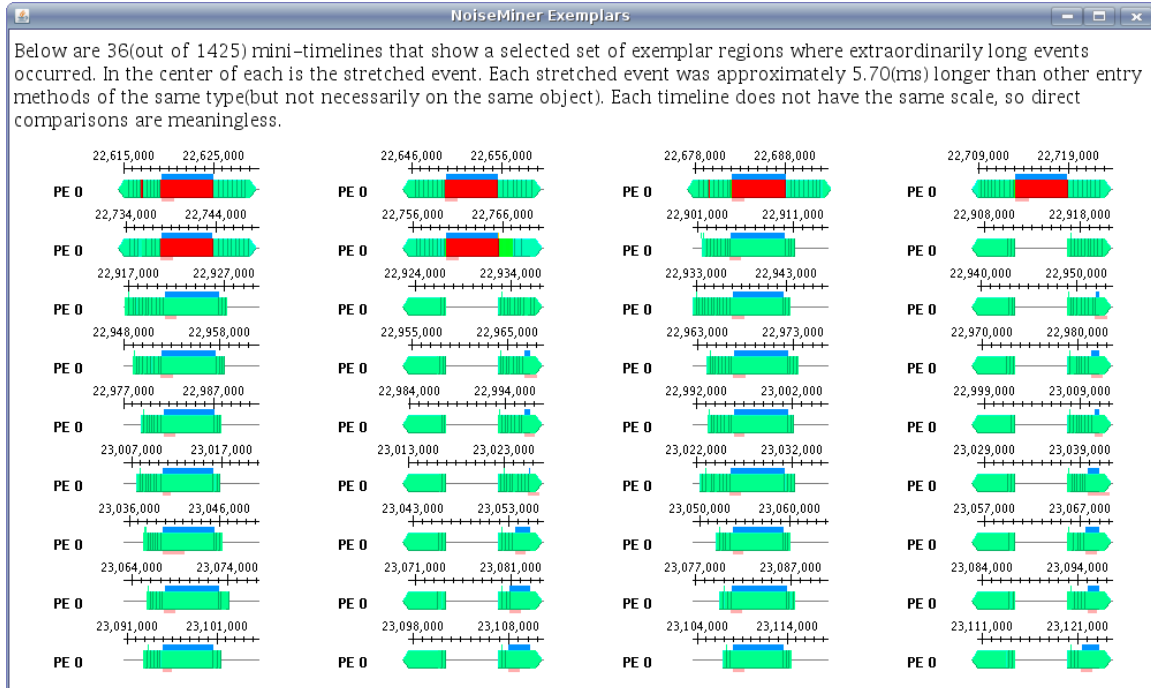
41: NoiseMiner View showing a 5.7 ms noise component that occurred 1425 times during a run. In this case, MPI calls to a faulty MPI implementation took an extra 5.7 ms to return.

The NoiseMiner view (see figure 41 and 42) displays statistics about abnormally long entry methods. Its purpose is to detect symptoms consistent with *Operating System Interference* or *Computational Noise* or *Software Interference*. The abnormally long events are filtered and clustered across multiple dimensions to produce a concise summary. The view displays both the duration of the events as well as the rate at which they occur. The initial dialog box allows a selection of processors and a time range. The user should select a time range that ignores any startup phase where events have chaotic durations. The tool makes only a single pass through the log files using a small bounded amount of memory, so the user should select as large time range as possible.

The tool uses stream mining techniques to produce its results by making only one pass through the input data while using a limited amount of memory. This allows NoiseMiner to be very fast and scalable.

The initial result window shows a list of zero or more noise components. Each noise component is a cluster of events whose durations are abnormally long. The noise duration for each event is computed by comparing the actual duration of the event with an expected duration of the event. Each noise component contains events of different types across one or more processors, but all the events within the noise component have similar noise durations.

Clicking on the "view" button for a noise component opens a window similar to figure 42. This second window displays up to 36 miniature timelines, each for a different event associated with the noise component.



42: NoiseMiner noise component view showing miniature timelines for one of the noise components.

NoiseMiner works by storing histograms of each entry method's duration. The histogram bins contain a window of recent occurrences as well as an average duration and count. After data stream has been parsed into the histogram bins, the histogram bins are clustered to determine the expected entry method duration. The histograms are then normalized by the expected duration so that they represent the abnormally stretched amounts for the entry methods. Then the histogram bins are clustered by duration and across processors. Any clusters that do not contribute much to the overall runtime are dropped.

15.2.4 Miscellaneous features

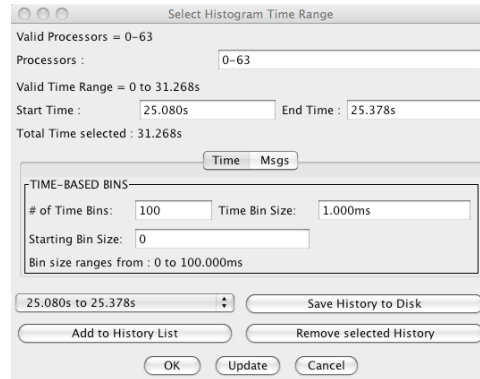
Standard Graph Display Interface

A standard graph display (an example of which can be found with the Main Summary Graph - figure 23) has the following features:

- **Graph types** can be selected between "Line Graph" which connects each data point with a colored line representing the appropriate data entry. This information may be "stacked" or "unstacked" (controlled by the checkbox to the right). A "stacked" graph places one data point set (Y values) on top of another. An "unstacked" graph simply uses the data point's Y value to directly determine the point's position; "Bar Graph" (the default) which draws a colored bar for each data entry and the value of the data point determines its height or starting position (depending on whether the bar graph is "stacked" or "unstacked"). A "Bar Graph" displayed in "unstacked" mode draws its bars in a tallest to shortest order so that the large Y values do not cover over the small Y values; "Area Graph" is similar to a "Line Graph" except that the area under the lines for a particular Y data point set is also colored by the data's appropriate color. "Area Graph"s are always stacked.
- **x-scale** allows the user to scale the X-Axis. This can be done by directly entering a scaling factor in the text field (simple numeric field - see below) or by using the "<<" or ">>" buttons to increase or decrease the scale by 0.25 each time. The "Reset" button changes the scale factor back to 1.0. A scrollbar automatically appears if the scale factor causes the canvas to be larger than the window.

- **y-scale** allows the user to scale the Y-Axis. This functions similarly to the **x-scale** feature where the buttons and fields are concerned.

Standard Dialog Features



43: An example Dialog with standard fields

Figure 43 shows a sample dialog box with standard features. The following are standard features that can be employed in such a dialog box:

- **Moving from field to field** via the tab key causes the dialog box update the last field input by the user. It also performs a consistency check. Whenever it finds an inconsistency, it will move mouse focus onto the offending field, disabling the “OK” button so as to force the user to fix the inconsistency. Examples of inconsistency includes: input that violates a field’s format; input whose value violates constraints (eg. start time larger than end time); or out-of-range stand-alone values.
- **Available buttons** include “OK” which confirms the user’s choice of parameters. This button is only activated if the dialog box considers the parameters’ input to be consistent. “Update” causes the dialog box to update the last field input by the user and perform a consistency check. This is similar in behavior to the user tabbing between fields. “Cancel” closes the dialog box without modifying any parameters if the tool has already been loaded or aborts the tool’s load attempt otherwise.
- **Parameter History** allows the user to quickly access information for all tools for a set of frequently needed time periods. An example of such a use is the desire by the analyst to view a particular phase or timestep of a computation without having to memorize or write on a piece of paper when exactly the phase or timestep occurred.

It consists of a pull-down text box and 2 buttons. “Add to History List” adds the current time range to the pull-down list to the left of the button. The dialog box maintains up to 5 entries, replacing older entries with newer ones. “Remove Selected History” removes the currently selected entry in the history list. “Save History to Disk” stores current history information to the file “ranges.hst” in the same directory where your logs are stored. Note that you will need write access to that directory to successfully store history information. A more flexible scheme is currently being developed and will be released in a later version of Projections. Clicking on the pull-down list allows the user to select one out of up to 5 possible time ranges. You can do so by moving the mouse up or down the list. Clicking on any one item changes the start and end times on the dialog box.

Data Fields

Throughout Projections tools and dialog boxes (see sample figure 43), data entry fields are provided. Unless otherwise specified, these can be of the following standard field with some format requirements:

- **Simple numeric fields:** An example can be found in figure 43 for “# of Time Bins:”. This field expects a single number.
- **Time-Based Field:** An example can be found in figure 43 for “Start Time:”. This field expects a single simple or floating point number followed by a time-scale modifier. The following modifiers are supported: *none* - this is the default and means the input number represents time in microseconds. A whole number is expected; *The characters “us”* - the input number represents time in microseconds. A whole number is expected; *The characters “ms”* - the input number represents time in milliseconds. This can be a whole number or floating point number; or *The character “s”* - the input number represents time in seconds. This can be a whole number or floating point number.
- **Processor-Based Field:** An example can be found in figure 43 for “Processors:”. This field expects a single whole number, a list of whole numbers, a range, or a mixed list of whole numbers and ranges. Here are some examples which makes the format clearer:

eg: Want to see processors 1, 3, 5, 7: Enter 1, 3, 5, 7

eg: Want to see processors 1, 2, 3, 4: Enter 1-4

eg: Want to see processors 1, 2, 3, 7: Enter 1-3, 7

eg: Want to see processors 1, 3, 4, 5, 7, 8: Enter 1, 3-5, 7-8

Ranges also allow skip-factors. Here are some examples:

eg: Want to see processors 3, 6, 9, 12, 15: Enter 3-15:3

eg: Want to see processors 1, 3, 6, 9, 11, 14: Enter 1, 3-9:3, 11, 14

This feature is extremely flexible. It will normalize your input to a canonical form, tolerating duplication of entries as well as out-of-order entries (ie. 4, 6, 3 is the same as 3-4, 6).

15.2.5 Known Issues

This section lists known issues and bugs with the Projections framework that we have not resolved at this time.

- Charm++ scheduler idle time is known to be incorrectly recorded on the BG/L machine at IBM TJ Watson.
- End-of-Run analysis techniques (while tracing applications) are currently known to hang for applications that make multiple calls to `traceBegin()` and `traceEnd()` on the same processor through multiple Charm++ objects.

Contents

- *Charm++ Debugger*
 - *Introduction*
 - *Building the Charm++ Debug Tool*
 - *Preparing the Charm++ Application for Debugging*
 - * *Record Replay*
 - *Running the Debugger*
 - * *CharmDebug command line parameters*
 - * *Basic usage*
 - * *Charm Debugging Related Options*
 - * *CharmDebug limitations*
 - * *Using the Debugger*
 - *Debugger Implementation Details*
 - * *Converse Client-Server Interface*

16.1 Introduction

The primary goal of the Charm++ parallel debugger is to provide an integrated debugging environment that allows the programmer to examine the changing state of parallel programs during the course of their execution.

The Charm++ debugging system has a number of useful features for Charm++ programmers. The system includes a Java GUI client program which runs on the programmer's desktop, and a Charm++ parallel program which acts as

a server. The client and server need not be on the same machine, and communicate over the network using a secure protocol described in *Converse Client-Server Interface*.

The system provides the following features:

- Provides a means to easily access and view the major programmer-visible entities, including array elements and messages in queues, across the parallel machine during program execution. Objects and messages are extracted as raw data, and interpreted by the debugger.
- Provides an interface to set and remove breakpoints on remote entry points, which capture the major programmer-visible control flows in a Charm++ program.
- Provides the ability to freeze and unfreeze the execution of selected processors of the parallel program, which allows a consistent snapshot by preventing things from changing as they are examined.
- Provides a way to attach a sequential debugger to a specific subset of processes of the parallel program during execution, which keeps a manageable number of sequential debugger windows open. Currently these windows are opened independently of the GUI interface, while in the future they will be transformed into an integrated view.

The debugging client provides these features via extensive support built into the Charm++ runtime.

16.2 Building the Charm++ Debug Tool

To get the CharmDebug tool, check out the source code from the following repository. This will create a directory named `ccs_tools`. Change to this directory and build the project.

```
$ git clone https://charm.cs.illinois.edu/gerrit/ccs_tools
$ cd ccs_tools
$ ant
```

This will create the executable `bin/charmdebug`.

You can also download the binaries from the Charm++ downloads website and use it directly without building. (NOTE: Binaries may work properly on some platforms, so building from the source code is recommended.)

16.3 Preparing the Charm++ Application for Debugging

Build Charm++ using `--enable-charmdebug` option. For example:

```
$ ./build charm++ netlrts-darwin-x86_64 --enable-charmdebug
```

No instrumentation is required to use the Charm++ debugger. Being CCS based, you can use it to set and step through entry point breakpoints and examine Charm++ structures in any Charm++ application.

Nevertheless, for some features to be present, some additional options may be required at either compile or link time:

- In order to provide a symbolic representation of the machine code executed by the application, the `-g` option is needed at compile time. This setting is needed to provide function names as well as source file names and line numbers wherever useful. This is also important to fully utilize `gdb` (or any other serial debugger) on one or more processes.
- Optimization options, by nature of transforming the source code, can produce a mismatch between the function displayed in the debugger (for example in a stack trace) and the functions present in the source code. To produce information coherent with source code, optimization is discouraged. Newer versions of some compilers support the `-Og` optimization level, which performs all optimizations that do not inhibit debugging.

- The link time option `-memory charmdebug` is only needed if you want to use the Memory view (see 16.4.5) or the Inspector framework (see 16.4.5) in CharmDebug.

16.3.1 Record Replay

The *Record Replay* feature is independent of the `charmdebug` application. It is a mechanism used to detect bugs that happen rarely depending on the order in which messages are processed. The program in consideration is first executed in record mode which produces a trace. When the program is run in replay mode it uses previously recorded trace to ensure that messages are processed in the same order as the recorded run. The idea is to make use of a message sequence number to satisfy a theorem says that the serial numbers will be the same if the messages are processed in the same order. .. *cite{rashmithesis}*

Record Replay tracing is automatically enabled for Charm++ programs and requires nothing special to be done during compilation. (Linking with the option `-tracemode recordreplay` used to be necessary). At run time, the `+record` option is used, which records messages in order in a file for each processor. The same execution order can be replayed using the `+replay` runtime option, which can be used at the same time as the other debugging tools in Charm++.

Note! If your Charm++ is built with `CMK_OPTIMIZE` on, all tracing will be disabled. So, use an unoptimized Charm++ to do your debugging.

16.4 Running the Debugger

16.4.1 CharmDebug command line parameters

- pes** Number of PEs
- +p** Number of PEs
- host** hostname of CCS server for application
- user** the username to use to connect to the hostname selected
- port** portnumber of CCS server for application
- sshtunnel** force the communication between client and server (in particular the one for CCS) to be tunnelled through ssh. This allow the bypass of firewalls.
- display** X Display

16.4.2 Basic usage

To run an application locally via the debugger on 4 PEs with command line options for your program (shown here as `opt1 opt2`):

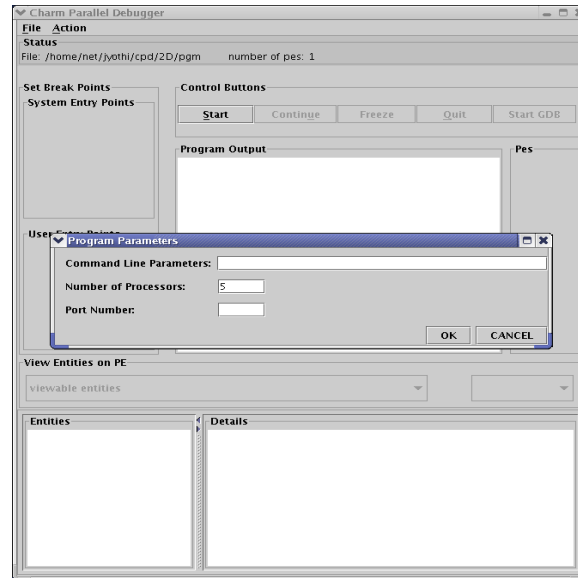
```
$ charmdebug pgm +p4 4 opt1 opt2
```

If the application should be run in a remote cluster behind a firewall, the previous command line will become:

```
$ charmdebug -host cluster.inst.edu -user myname -sshtunnel pgm +p4 4 opt1 opt2
```

CharmDebug can also be executed without any parameters. The user can then choose the application to launch and its command line parameters from within the **File** menu as shown in Figure 44.

Note: `charmdebug` command line launching only works on `netlrts-*` and `verbs-*` builds of Charm++.



44: Using the menu to set parameters for the Charm++ program

To replay a previously recorded session:

```
$ charmdebug pgm +p4 opt1 opt2 +replay
```

16.4.3 Charm Debugging Related Options

When using the Charm debugger to launch your application, it will automatically set these to defaults appropriate for most situations.

+cpd Triggers application freeze at startup for debugger.

++charmdebug Triggers charmrund to provide some information about the executable, as well as provide an interface to gdb for querying.

+debugger Which debuggers to use.

++debug Run each node under gdb in an xterm window, prompting the user to begin execution.

++debug-no-pause Run each node under gdb in an xterm window immediately (i.e. without prompting the user to begin execution).

Note: If you're using the charm debugger it will probably be best to control the sequential (i.e. gdb) debuggers from within its GUI interface.

++DebugDisplay X Display for xterm

++server-port Port to listen for CCS requests

++server Enable client-server (CCS) mode

+record Use the recordreplay tracemode to record the exact event/message sequence for later use.

+replay Force the use of recorded log of events/messages to exactly reproduce a previous run.

The preceding pair of commands **+record +replay** are used to produce the “instant replay” feature. This feature is valuable for catching errors which only occur sporadically. Such bugs which arise from the nondeterminacy of parallel execution can be fiendishly difficult to replicate in a debugging environment. Typical usage is

to keep running the application with `+record` until the bug occurs. Then run the application under the debugger with the `+replay` option.

16.4.4 CharmDebug limitations

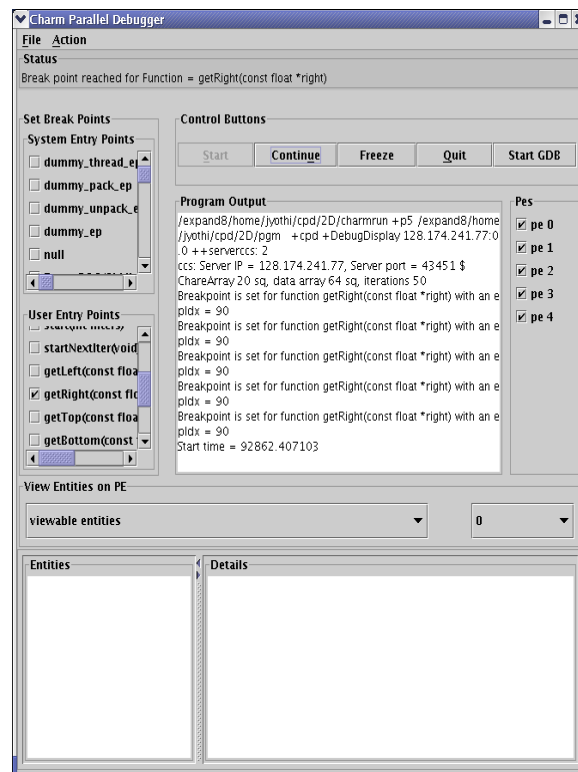
Clusters

CharmDebug is currently limited to applications started directly by the debugger due to implementation peculiarities. It will be extended to support connection to remote running applications in the near future.

Due to the current implementation, the debugging tool is limited to `netlrts-*` and `verbs-*` versions. Other builds of Charm++ might have unexpected behavior. In the near future this will be extended at least to the `mpi-*` versions.

Record Replay

The *Record Replay* feature does not work well with spontaneous events. Load balancing is the most common form of spontaneous event in that it occurs periodically with no other causal event.



45: Parallel debugger when a break point is reached

As per Rashmi's thesis:

“There are some unique issues for replay in the context of Charm because it provides high-level support for dynamic load balancing, quiescence detection and information sharing. Many of the load balancing strategies in Charm have a spontaneous component. The strategy periodically checks the sizes of the queues on the local processor. A replay load balancing strategy implements the known load redistribution. The behavior of the old balancing strategy is therefore not replayed only its effect is. Since minimal

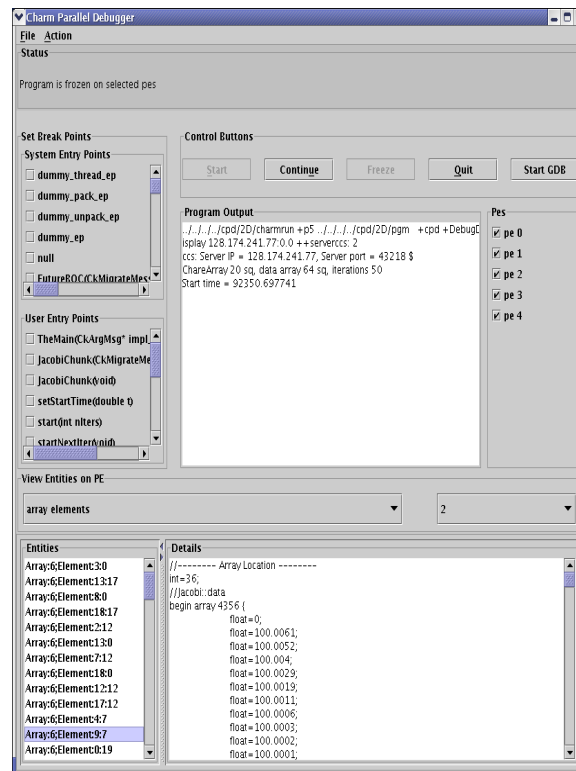
tracing is used by the replay mechanism the amount of perturbation due to tracing is reduced. The replay mechanism is proposed as a debugging support to replay asynchronous message arrival orders.”

Moreover, if your application crashes without a clean shutdown, the log may be lost with the application.

16.4.5 Using the Debugger

Once the debugger’s GUI loads, the programmer triggers the program execution by clicking the *Start* button. When starting by command line, the application is automatically started. The program begins by displaying the user and system entry points as a list of check boxes, pausing at the onset. The user could choose to set breakpoints by clicking on the corresponding entry points and kick off execution by clicking the *Continue* Button. Figure 45 shows a snapshot of the debugger when a breakpoint is reached. The program freezes when a breakpoint is reached.

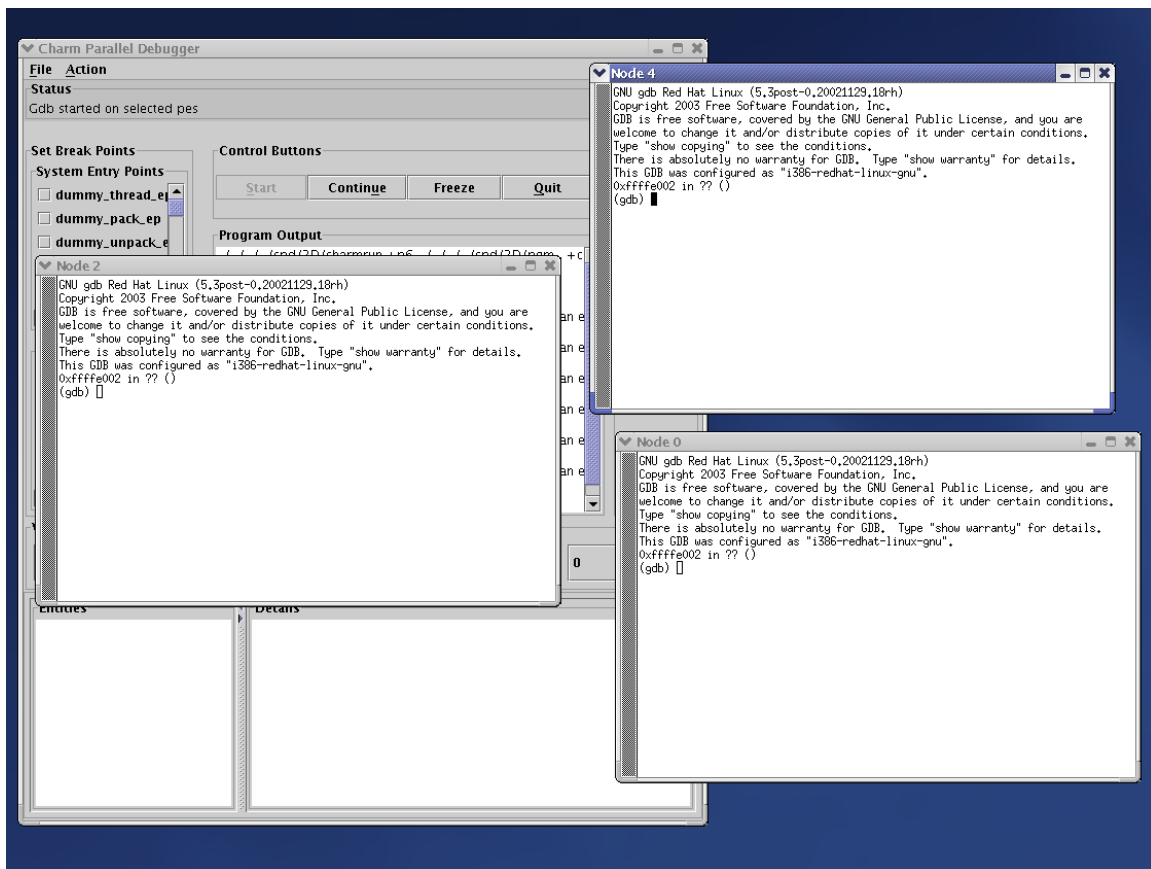
Clicking the *Freeze* button during the execution of the program freezes execution, while *Continue* button resumes execution. The *Quit* button can be used to abort execution at any point of time. Entities (for instance, array elements) and their contents on any processor can be viewed at any point in time during execution as illustrated in Figure 46.



46: Freezing program execution and viewing the contents of an array element using the Parallel Debugger

Specific individual processes of the Charm++ program can be attached to instances of *gdb* as shown in Figure 47. The programmer chooses which PEs to connect *gdb* processes to via the checkboxes on the right side. *Note!* While the program is suspended in *gdb* for step debugging, high-level CharmDebug features such as object inspection will not work.

Charm++ objects can be examined via the *View Entities on PE : Display* selector. It allows the user to choose from *Charm Objects*, *Array Elements*, *Messages in Queue*, *Readonly Variables*, *Readonly Messages*, *Entry Points*, *Chare Types*, *Message Types* and *Mainchares*. The right sideselector sets the PE upon which the request for display will be made. The user may then click on the *Entity* to see the details.



47: Parallel debugger showing instances of *gdb* open for the selected processor elements

Memory View

The menu option Action → Memory allows the user to display the entire memory layout of a specific processor. An example is shown in Figure 48. This layout is colored and the colors have the following meaning:

red memory allocated by the Charm++ Runtime System;

blue memory allocated directly by the user in its code;

pink memory used by messages;

orange memory allocated to a chare element;

black memory not allocated;

gray a big jump in memory addresses due to the memory pooling system, it represent a large portion of virtual space not used between two different zones of used virtual space address;

yellow the currently selected memory slot;

Currently it is not possible to change this color association. The bottom part of the view shows the stack trace at the moment when the highlighted (yellow) memory slot was allocated. By left clicking on a particular slot, this slot is fixed in highlight mode. This allows a more accurate inspection of its stack trace when this is large and does not fit the window.

Info → Show Statistics will display a small information box like the one in Figure 49.

A useful tool of this view is the memory leak search. This is located in the menu Action → Search Leaks. The processor under inspection runs a reachability test on every memory slot allocated to find if there is a pointer to it. If there is none, the slot is partially colored in green, to indicate its status of leak. The user can then inspect further these slots. Figure 50 shows some leaks being detected.

If the memory window is kept open while the application is unfrozen and makes progress, the loaded image will become obsolete. To cope with this, the “Update” button will refresh the view to the current allocation status. All the leaks that had been already found as such, will still be partially colored in green, while the newly allocated slots will not, even if leaking. To update the leak status, re-run the Search Leaks tool.

Finally, when a specific slot is highlighted, the menu Action → Inspect opens a new window displaying the content of the memory in that slot, as interpreted by the debugger (see next subsection for more details on this).

Inspector framework

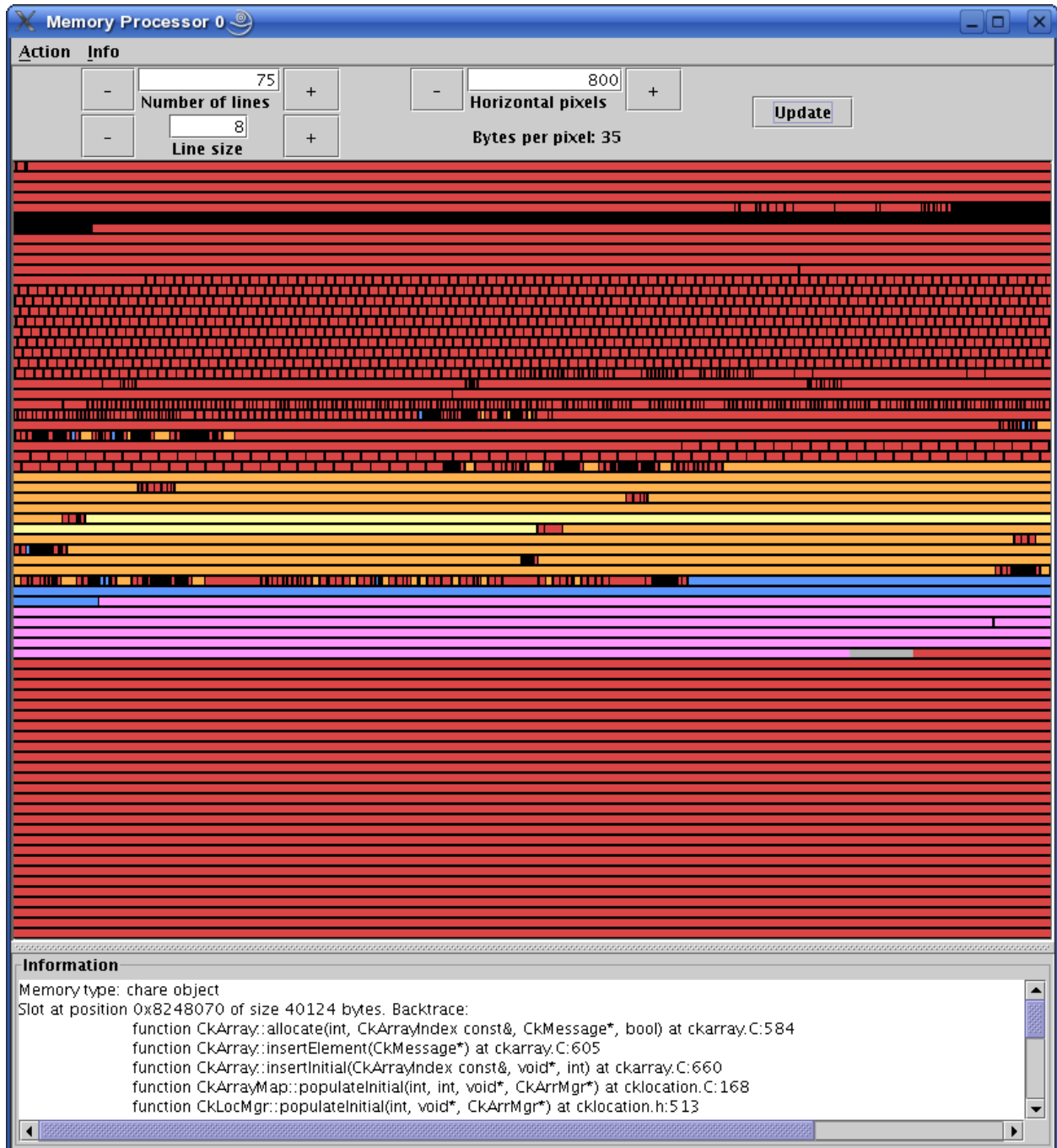
Without any code rewriting of the application, CharmDebug is capable of loading a raw area of memory and parsing it with a given type name. The result (as shown in Figure 51, is a browsable tree. The initial type of a memory area is given by its virtual table pointer (Charm++ objects are virtual and therefore loadable). In the case of memory slots not containing classes with virtual methods, no display will be possible.

When the view is open and is displaying a type, by right clicking on a leaf containing a pointer to another memory location, a popup menu will allow the user to ask for its dereference (shown in Figure 51). In this case, CharmDebug will load this raw data as well and parse it with the given type name of the pointer. This dereference will be inlined and the leaf will become an internal node of the browse tree.

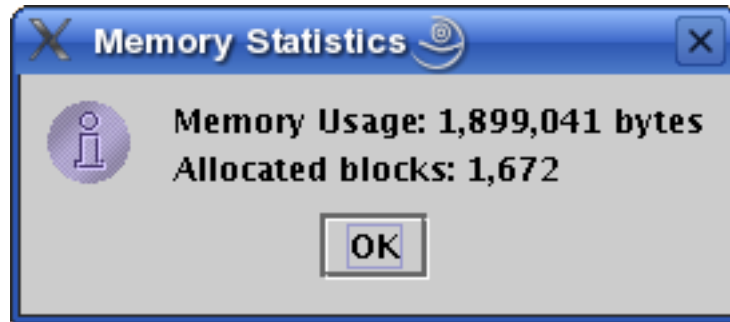
16.5 Debugger Implementation Details

The following classes in the PUP framework were used in implementing debugging support in charm.

- `class PUP::er` - This class is the abstract superclass of all the other classes in the framework. The `pup` method of a particular class takes a reference to a `PUP::er` as parameter. This class has methods for dealing



48: Main memory view



49: Information box display memory statistics

with all the basic C++ data types. All these methods are expressed in terms of a generic pure virtual method. Subclasses only need to provide the generic method.

- `class PUP::toText` - This is a subclass of the `PUP::toTextUtil` class which is a subclass of the `PUP::er` class. It copies the data of an object to a C string, including the terminating NULL.
- `class PUP::sizerText` - This is a subclass of the `PUP::toTextUtil` class which is a subclass of the `PUP::er` class. It returns the number of characters including the terminating NULL and is used by the `PUP::toText` object to allocate space for building the C string.

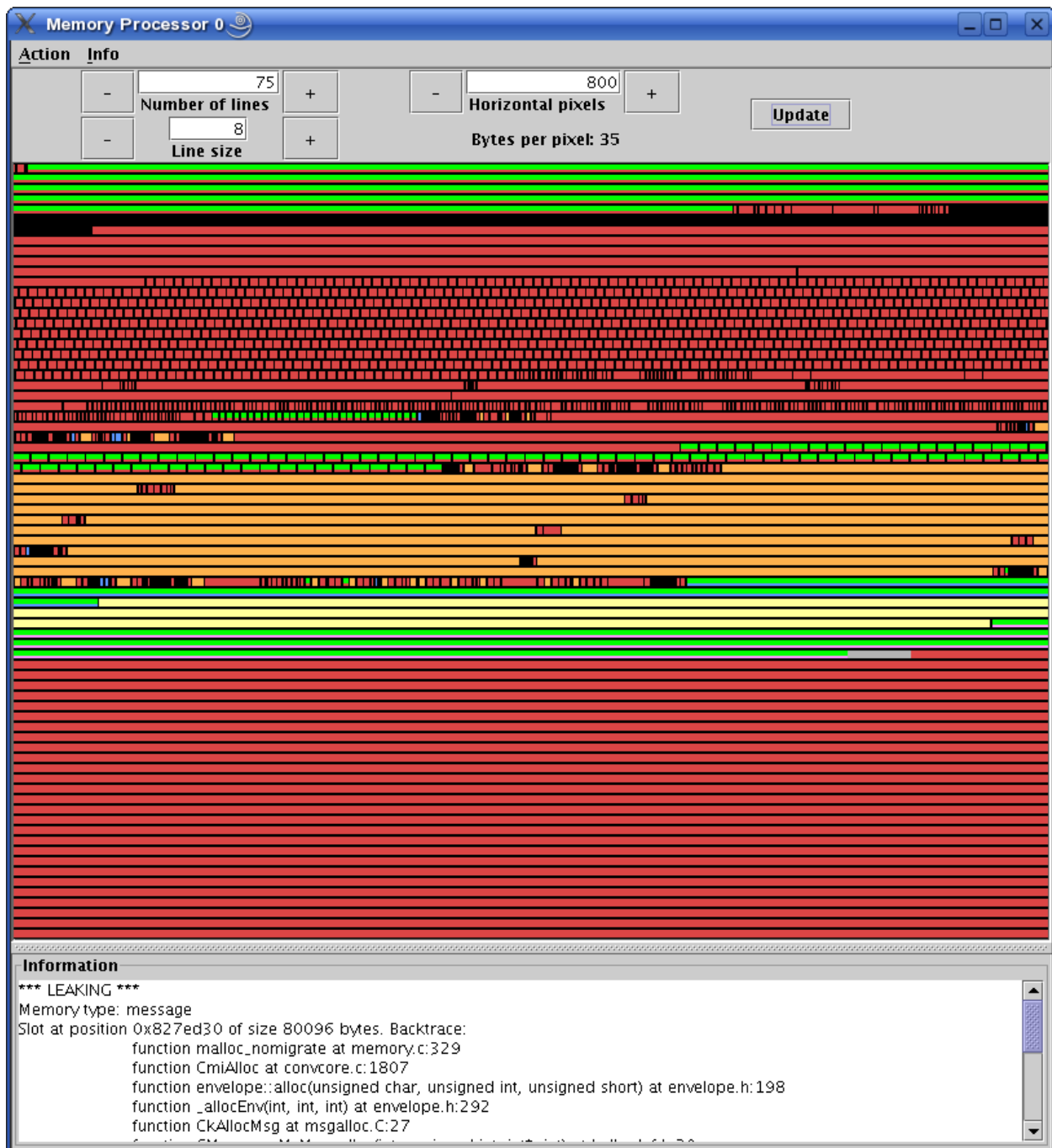
The code below shows a simple class declaration that includes a `pup` method.

```
class foo {
private:
    bool isBar;
    int x;
    char y;
    unsigned long z;
    float q[3];
public:
    void pup(PUP::er &p) {
        p(isBar);
        p(x); p(y); p(z);
        p(q, 3);
    }
};
```

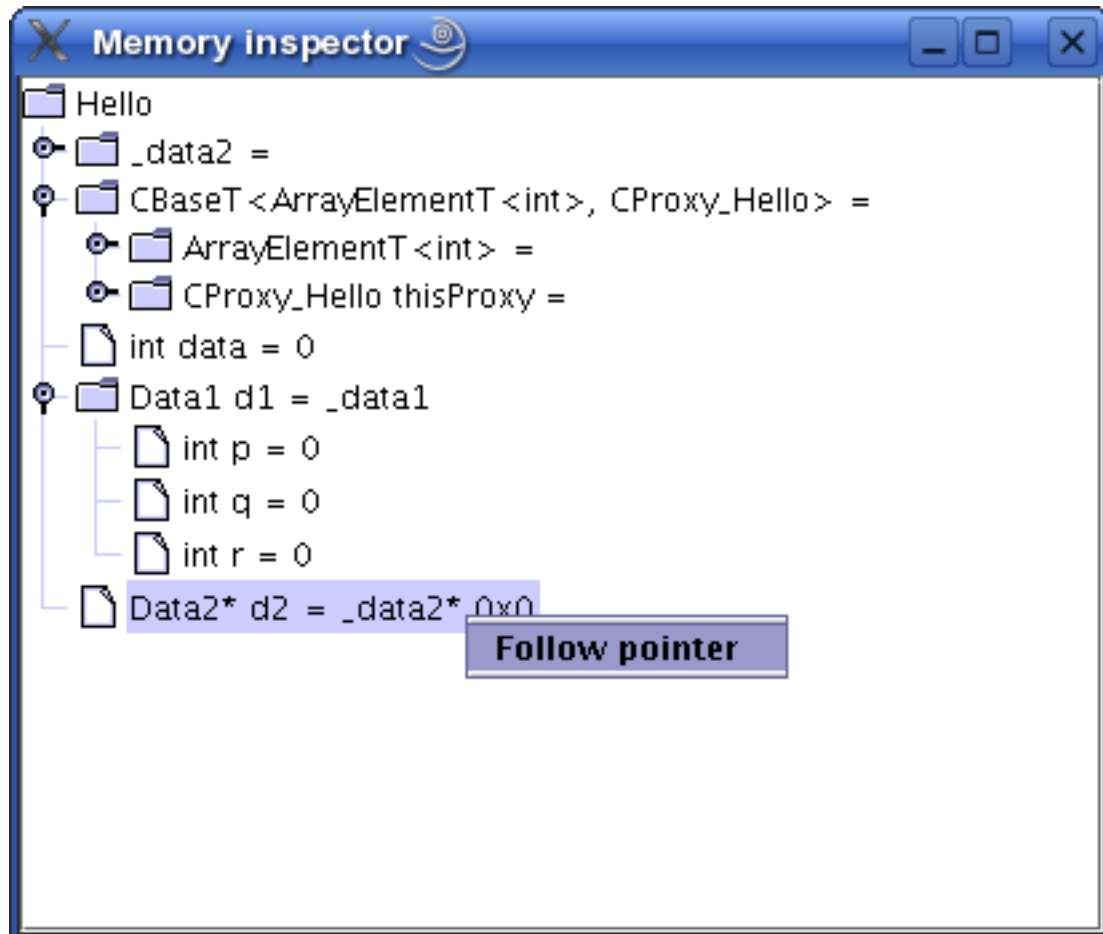
16.5.1 Converse Client-Server Interface

The Converse Client-Server (CCS) module enables Converse .. *cite{InterOpIPPS96}* programs to act as parallel servers, responding to requests from non-Converse programs. The CCS module is split into two parts – client and server. The server side is used by Converse programs while the client side is used by arbitrary non-Converse programs. A CCS client accesses a running Converse program by talking to a server-host which receives the CCS requests and relays them to the appropriate processor. The server-host is `charmrun .. cite{charmman}` for netlrts-versions and is the first processor for all other versions.

In the case of the netlrts- version of Charm++, a Converse program is started as a server by running the Charm++ program using the additional runtime option `++server`. This opens the CCS server on any TCP port number. The TCP port number can be specified using the command-line option `server-port`. A CCS client connects to a CCS server, asks a server PE to execute a pre-registered handler and receives the response data. The function `CcsConnect` takes a pointer to a `CcsServer` as an argument and connects to the given CCS server. The functions `CcsNumNodes`, `CcsNumPes`, and `CcsNodeSize` implemented as part of the client interface in Charm++ return information about



50: Memory view after running the Search Leaks tool



51: Raw memory parsed and displayed as a tree

the parallel machine. The function `CcsSendRequest` takes a handler ID and the destination processor number as arguments and asks the server to execute the requested handler on the specified processor. `CcsRecvResponse` receives a response to the previous request in-place. A timeout is also specified which gives the number of seconds to wait until the function returns 0, otherwise the number of bytes received is returned.

Once a request arrives on a CCS server socket, the CCS server runtime looks up the appropriate registered handler and calls it. If no handler is found the runtime prints a diagnostic and ignores the message. If the CCS module is disabled in the core, all CCS routines become macros returning 0. The function `CcsRegisterHandler` is used to register handlers in the CCS server. A handler ID string and a function pointer are passed as parameters. A table of strings corresponding to appropriate function pointers is created. Various built-in functions are provided which can be called from within a CCS handler. The debugger behaves as a CCS client invoking appropriate handlers which make use of some of these functions. Some of the built-in functions are as follows.

- `CcsSendReply` - This function sends the data provided as an argument back to the client as a reply. This function can only be called from a CCS handler invoked remotely.
- `CcsDelayReply` - This call is made to allow a CCS reply to be delayed until after the handler has completed.

The CCS runtime system provides several built-in CCS handlers, which are available to any Converse program. All Charm++ programs are essentially Converse programs. `ccs_getinfo` takes an empty message and responds with information about the parallel job. Similarly the handler `ccs_killport` allows a client to be notified when a parallel run exits.

Contents

- *Charisma*
 - *Introduction*
 - *Charisma Syntax*
 - * *Orchestration Code*
 - * *Sequential Code*
 - *Building and Running a Charisma Program*
 - *Support for Library Module*
 - *Writing Module Library*
 - *Using Module Library*
 - *Using Load Balancing Module*
 - * *Coding*
 - * *Compiling and Running*
 - *Handling Sparse Object Arrays*
 - *Example: Jacobi 1D*

17.1 Introduction

This manual describes Charisma, an orchestration language for migratable parallel objects. Charisma can be downloaded from <https://charm.cs.illinois.edu/gerrit/gitweb?p=Charisma.git>

17.2 Charisma Syntax

A Charisma program is composed of two parts: the *orchestration* code in a **.or** file, and sequential user code in C/C++ form.

17.2.1 Orchestration Code

The orchestration code in the **.or** file can be divided into two parts. The header part contains information about the program, included external files, defines, and declaration of parallel constructs used in the code. The orchestration section is made up of statements that form a global control flow of the parallel program. In the orchestration code, Charisma employs a macro dataflow approach; the statements produce and consume values, from which the control flows can be organized, and messages and method invocations generated.

Header Section

The very first line should give the name of the Charisma program with the `program` keyword.

```
program jacobi
```

The `program` keyword can be replaced with `module`, which means that the output program is going to be a library module instead of a stand-alone program. Please refer to Section 17.4 for more details.

Next, the programmer can include external code files in the generated code with the keyword `include` specifying the filename without extension. For example, the following statement tells the Charisma compiler to look for header file “particles.h” to be included in the generated header file “jacobi.h” and to look for C/C++ code file “particles.[C | cc | cpp | cxx | c]” to be included in the generated C++ code file “jacobi.C”.

```
include particles;
```

It is useful when there is source code that must precede the generated parallel code, such as basic data structure declaration.

After the `include` section is the `define` section, where environmental variables can be defined for Charisma. For example, to tell Charisma to generate additional code to enable the load balancing module, the programmer needs to use `define ldb;` in the orchestration code. Please refer to Section 17.7 for details.

Declaration Section

Next comes the declaration section, where classes, objects and parameters are declared. A Charisma program is composed of multiple sets of parallel objects which are organized by the orchestration code. Different sets of objects can be instantiated from different class types. Therefore, we have to specify the class types and object instantiation. Also we need to specify the parameters (See Section 17.2.1) to use in the orchestration statements.

A Charisma program or module has one “MainChare” class, and it does not require explicit instantiation since it is a singleton. The statement to declare MainChare looks like this:

```
class JacobiMain : MainChare;
```

For object arrays, we first need to declare the class types inherited from 1D object array, 2D object array, etc, and then instantiate from the class types. The dimensionality information of the object array is given in a pair of brackets with each dimension size separated by a comma.


```
class JacobiWorker : ChareArray1D;
obj workers : JacobiWorker[N];

class Cell : ChareArray3D;
obj cells : Cell[M,M,M];
```

Note that key word `class` is for class type derivation, and `obj` is for parallel object or object array instantiation. The above code segment declares a new class type `JacobiWorker` which is a 1D object array, and the programmer is supposed to supply sequential code for it in files `JacobiWorker.h` and `JacobiWorker.C` (see Section 17.2.2 for more details on sequential code). Object array `workers` is instantiated from `JacobiWorker` and has 16 elements.

The last part is orchestration parameter declaration. These parameters are used only in the orchestration code to connect input and output of orchestration statements, and their data type and size is declared here. More explanation of these parameters can be found in Section 17.2.1.

```
param lb : double[N];
param rb : double[N];
```

With this, `lb` and `rb` are declared as parameters that can be “connected” with local variables of double array of size of 512.

Orchestration Section

In the main body of orchestration code, the programmer describes the behavior and interaction of the elements of the object arrays using orchestration statements.

• Foreach Statement

The most common kind of parallelism is the invocation of a method across all elements in an object array. Charisma provides a `foreach` statement for specifying such parallelism. The keywords `foreach` and `end-foreach` forms an enclosure within which the parallel invocation is performed. The following code segment invokes the entry method `compute` on all the elements of array `myWorkers`.

```
foreach i in workers
  workers[i].compute();
end-foreach
```

• Publish Statement and Produced/Consumed Parameters

In the orchestration code, an object method invocation can have input and output (consumed and produced) parameters. Here is an orchestration statement that exemplifies the input and output of this object methods `workers.produceBorders` and `workers.compute`.

```
foreach i in workers
  (lb[i], rb[i]) <- workers[i].produceBorders();
  workers[i].compute(lb[i+1], rb[i-1]);

  (+error) <- workers[i].reduceData();
end-foreach
```

Here, the entry method `workers[i].produceBorders` produces (called *published* in Charisma) values of `lb[i]`, `rb[i]`, enclosed in a pair of parentheses before the publishing sign `<-`. In the second statement, function `workers[i].compute` consumes values of `lb[i+1]`, `rb[i-1]`, just like normal function parameters. If a reduction operation is needed, the reduced parameter is marked with a `+` before it, like the `error` in the third statement.

An entry method can have arbitrary number of published (produced and reduced) values and consumed values. In addition to basic data types, each of these values can also be an object of arbitrary type. The values published by `A[i]` must have the index `i`, whereas values consumed can have the index `e(i)`, which is an index expression in the form of `i±c` where `c` is a constant. Although we have used different symbols (`p` and `q`) for the input and the output variables, they are allowed to overlap.

The parameters are produced and consumed in the program order. Namely, a parameter produced in an early statement will be consumed by the next consuming statement, but will no longer be visible to any consuming statement after a subsequent statement producing the same parameter in program order. Special rules involving loops are discussed later with loop statement.

• Overlap Statement

Complicated parallel programs usually have concurrent flows of control. To explicitly express this, Charisma provides a `overlap` keyword, whereby the programmer can fire multiple overlapping control flows. These flows may contain different number of steps or statements, and their execution should be independent of one another so that their progress can interleave with arbitrary order and always return correct results.

```
overlap
{
  foreach i in workers1
    (lb[i], rb[i]) <- workers1[i].produceBorders();
  end-foreach
  foreach i in workers1
    workers1[i].compute(lb[i+1], rb[i-1]);
  end-foreach
}
{
  foreach i in workers2
    (lb[i], rb[i]) <- workers2[i].compute(lb[i+1], rb[i-1]);
  end-foreach
}
end-overlap
```

This example shows an `overlap` statement where two blocks in curly brackets are executed in parallel. Their execution joins back to one at the end mark of `end-overlap`.

• Loop Statement

Loops are supported with `for` statement and `while` statement. Here are two examples.

```
for iter = 0 to MAX_ITER
  workers.doWork();
end-for
```

```
while (err > epsilon)
  (+err) <- workers.doWork();
  MainChare.updateError(err);
end-while
```

The loop condition in `for` statement is independent from the main program; it simply tells the program to repeat the block for so many times. The loop condition in `while` statement is actually updated in the `MainChare`. In the above example, `err` and `epsilon` are both member variables of class `MainChare`, and can be updated as the example shows. The programmer can activate the “autoScalar” feature by including a `define autoScalar;` statement in the orchestration code. When `autoScalar` is enabled, Charisma will find all the scalars in the `.or` file, and create a local copy in the `MainChare`. Then every time the scalar is published by a statement, an update statement will automatically be inserted after that statement. The only thing that the programmer needs to do is to initialize the local scalar with a proper value.

Rules of connecting produced and consumed parameters concerning loops are natural. The first consuming statement will look for values produced by the last producing statement before the loop, for the first iteration. The last producing statement within the loop body, for the following iterations. At the last iteration, the last produced values will be disseminated to the code segment following the loop body. Within the loop body, program order holds.

```
for iter = 1 to MAX_ITER
  foreach i in workers
    (lb[i], rb[i]) <- workers[i].compute(lb[i+1], rb[i-1]);
  end-foreach
end-for
```

One special case is when one statement's produced parameter and consumed parameter overlaps. It must be noted that there is no dependency within the same `foreach` statement. In the above code segment, the values consumed `lb[i]`, `rb[i]` by `worker[i]` will not come from its neighbors in this iteration. The rule is that the consumed values always originate from previous `foreach` statements or `foreach` statements from a previous loop iteration, and the published values are visible only to following `foreach` statements or `foreach` statements in following loop iterations.

• Scatter and Gather Operation

A collection of values produced by one object may be split and consumed by multiple object array elements for a scatter operation. Conversely, a collection of values from different objects can be gathered to be consumed by one object.

```
foreach i in A
  (points[i,*]) <- A[i].f(...);
end-foreach
foreach k,j in B
  (...) <- B[k,j].g(points[k,j]);
end-foreach
```

A wildcard dimension `*` in `A[i].f()`'s output `points` specifies that it will publish multiple data items. At the consuming side, each `B[k,j]` consumes only one point in the data, and therefore a scatter communication will be generated from `A` to `B`. For instance, `A[1]` will publish data `points[1,0..N-1]` to be consumed by multiple array objects `B[1,0..N-1]`.

```
foreach i,j in A
  (points[i,j]) <- A[i,j].f(...);
end-foreach
foreach k in B
  (...) <- B[k].g(points[*,k]);
end-foreach
```

Similar to the scatter example, if a wildcard dimension `*` is in the consumed parameter and the corresponding published parameter does not have a wildcard dimension, there is a gather operation generated from the publishing statement to the consuming statement. In the following code segment, each `A[i,j]` publishes a data point, then data points from `A[0..N-1,j]` are combined together to for the data to be consumed by `B[j]`.

Many communication patterns can be expressed with combination of orchestration statements. For more details, please refer to PPL technical report 06-18, "Charisma: Orchestrating Migratable Parallel Objects".

Last but not least, all the orchestration statements in the `.or` file together form the dependency graph. According to this dependency graph, the messages are created and the parallel program progresses. Therefore, the user is advised to put only parallel constructs that are driven by the data dependency into the orchestration code. Other elements such as local dependency should be coded in the sequential code.

17.2.2 Sequential Code

Sequential Files

The programmer supplies the sequential code for each class as necessary. The files should be named in the form of class name with appropriate file extension. The header file is not really an ANSI C header file. Instead, it is the sequential portion of the class's declaration. Charisma will generate the class declaration from the orchestration code, and incorporate the sequential portion in the final header file. For example, if a molecular dynamics simulation has the following classes (as declared in the orchestration code):

```
class MDMain : MainChare;
class Cell : ChareArray3D;
class CellPair : ChareArray6D;
```

The user is supposed to prepare the following sequential files for the classes: MDMain.h, MDMain.C, Cell.h, Cell.C, CellPair.h and CellPair.C, unless a class does not need sequential declaration and/or definition code. Please refer to the example in the Appendix.

For each class, a member function `void initialize(void)` can be defined and the generated constructor will automatically call it. This saves the trouble of explicitly call initialization code for each array object.

Producing and Consuming Functions

The C/C++ source code is nothing different than ordinary sequential source code, except for the producing/consuming part. For consumed parameters, a function treats them just like normal parameters passed in. To handle produced parameters, the sequential code needs to do two special things. First, the function should have an extra parameter for output parameters. The parameter type is keyword `outport`, and the parameter name is the same as appeared in the orchestration code. Second, in the body of the function, the keyword `produce` is used to connect the orchestration parameter and the local variables whose value will be sent out, in a format of a function call, as follows.

```
produce(produced_parameter, local_variable[, size_of_array]);
```

When the parameter represents a data array, we need the additional `size_of_array` to specify the size of the data array.

The dimensionality of an orchestration parameter is divided into two parts: its dimension in the orchestration code, which is implied by the dimensionality of the object arrays the parameter is associated, and the local dimensionality, which is declared in the declaration section. The orchestration dimension is not explicitly declared anywhere, but it is derived from the object arrays. For instance, in the 1D Jacobi worker example, `lb` and `rb` has the same orchestration dimensionality of workers, namely 1D of size 16. The local dimensionality is used when the parameter is associated with local variables in sequential code. Since `lb` and `rb` are declared to have the local type and dimension of `double [512]`, the producing statement should connect it with a local variable of `double [512]`.

```
void JacobiWorker::produceBorders(outport lb, outport rb) {
    ...
    produce(lb, localLB, 512);
    produce(rb, localRB, 512);
}
```

Special cases of the produced/consumed parameters involve scatter/gather operations. In scatter operation, since an additional dimension is implied in the produced parameter, we the `local_variable` should have additional dimension equal to the dimension over which the scatter is performed. Similarly, the input parameter in gather operation will have an additional dimension the same size of the dimension of the gather operation.

For reduction, one additional parameter of type `char []` is added to specify the reduction operation. Built-in reduction operations are `+` (sum), `*` (product), `<` (minimum), `>` (maximum) for basic data types. For instance the following

statements takes the sum of all local value of `result` and for output in `sum`.

```
reduce(sum, result, "+");
```

If the data type is a user-defined class, then you might use the function or operator defined to do the reduction. For example, assume we have a class called `Force`, and we have an `add` function (or a `+` operator) defined.

```
Force& Force::add(const Force& f);
```

In the reduction to sum all the local forces, we can use

```
reduce(sumForces, localForce, "add");
```

Miscellaneous Issues

In sequential code, the user can access the object's index by a keyword `thisIndex`. The index of 1-D to 6-D object arrays are:

```
1D: thisIndex
2D: thisIndex.{x,y}
3D: thisIndex.{x,y,z}
4D: thisIndex.{w,x,y,z}
5D: thisIndex.{v,w,x,y,z}
6D: thisIndex.{x1,y1,z1,x2,y2,z2}
```

17.3 Building and Running a Charisma Program

There are two steps to build a Charisma program: generating Charm++ program from orchestration code, and building the Charm++ program.

1) Charisma compiler, currently named `orchc`, is used to compile the orchestration code (`.or` file) and integrate sequential code to generate a Charm++ program. The resultant Charm++ program usually consists of the following code files: Charm++ Interface file (`[modulename].ci`), header file (`[modulename].h`) and C++ source code file (`[modulename].C`). The command for this step is as follows.

```
$ orchc [modulename].or
```

2) Charm++ compiler, `charmc`, is used to parse the Charm++ Interface (`.ci`) file, compile C/C++ code, and link and build the executable. The typical commands are:

```
$ charmc [modulename].ci
$ charmc [modulename].C -c
$ charmc [modulename].o -o pgm -language charm++
```

Running the Charisma program is the same as running a Charm++ program, using Charm++'s job launcher `charmrun` (on some platforms like CSE's Turing Cluster, use the customized job launcher `rjq` or `rj`).

```
$ charmrun pgm +p4
```

Please refer to Charm++'s manual and tutorial for more details of building and running a Charm++ program.

17.4 Support for Library Module

Charisma is capable of producing library code for reuse with another Charisma program. We explain this feature in the following section.

17.5 Writing Module Library

The programmer uses the keyword `module` instead of `program` in the header section of the orchestration code to tell the compiler that it is a library module. Following keyword `module` is the module name, then followed by a set of configuration variables in a pair parentheses. The configuration variables are used in creating instances of the library, for such info as problem size.

Following the first line, the library's input and output parameters are posted with keywords `inparam` and `outparam`.

```
module FFT3D (CHUNK, M, N);  
inparam indata;  
outparam outdata1, outdata2;
```

The body of the library is not very different from that of a normal program. It takes input parameters and produces output parameters, as posted in the header section.

17.6 Using Module Library

To use a Charisma module library, the programmer first needs to create an instance of the library. There are two steps: including the module and creating an instance.

```
use FFT3D;  
library f1 : FFT3D (CHUNK=10, M=10, N=100);  
library f2 : FFT3D (CHUNK=8, M=8, N=64);
```

The keyword `use` and the module name includes the module in the program, and the keyword `library` creates an instance with the instance name, followed by the module name with value assignment of configuration variables. These statements must appear in the declaration section before the library instance can be used in the main program's orchestration code.

Invoking the library is like calling a publish statement; the input and output parameters are the same, and the object name and function name are replaced with the library instance name and the keyword `call` connected with a colon.

```
(f1_outdata[*]) <- f1:call(f1_indata[*]);
```

Multiple instances can be created out of the same module. Their execution can interleave without interfering with one another.

17.7 Using Load Balancing Module

17.7.1 Coding

To activate the load balancing module and prepare objects for migration, there are 3 things that need to be added in Charisma code.

First, the programmer needs to inform Charisma about load balancing with a `define ldb;` statement in the header section of the orchestration code. This will make Charisma generate extra Charm++ code to do load balancing such as PUP methods.

Second, the user has to provide a PUP function for each class with sequential data that needs to be moved when the object migrates. When choosing which data items to pup, the user has the flexibility to leave the dead data behind to save on communication overhead in migration. The syntax for the sequential PUP is similar to that in a Charm++ program. Please refer to the load balancing section in Charm++ manual for more information on PUP functions. A typical example would look like this in user's sequential .C file:

```
void JacobiWorker::sequentialPup(PUP::er& p){
    p|myLeft; p|myRight; p|myUpper; p|myLower;
    p|myIter;
    PUPArray(p, (double *)localData, 1000);
}
```

Thirdly, the user will make the call to invoke load balancing session in the orchestration code. The call is `AtSync()`; and it is invoked on all elements in an object array. The following example shows how to invoke load balancing session every 4th iteration in a for-loop.

```
for iter = 1 to 100
    // work work
    if (iter % 4 == 0) then
        foreach i in workers
            workers[i].AtSync();
        end-foreach
    end-if
end-for
```

If a while-loop is used instead of for-loop, then the test-condition in the `if` statement is a local variable in the program's `MainChare`. In the sequential code, the user can maintain a local variable called `iter` in `MainChare` and increment it every iteration.

17.7.2 Compiling and Running

Unless linked with load balancer modules, a Charisma program will not perform actual load balancing. The way to link in a load balancer module is adding `-module EveryLB` as a link-time option.

At run-time, the load balancer is specified in command line after the `+balancer` option. If the balancer name is incorrect, the job launcher will automatically print out all available load balancers. For instance, the following command uses `RefineLB`.

```
$ ./charmrun ./pgm +p16 +balancer RefineLB
```

17.8 Handling Sparse Object Arrays

In Charisma, when we declare an object array, by default a dense array is created with all the elements populated. For instance, when we have the following declaration in the orchestration code, an array of $N \times N \times N$ is created.

```
class Cell : ChareArray3D;
obj cells : Cell[N,N,N];
```

There are certain occasions when the programmer may need sparse object arrays, in which not all elements are created. An example is neighborhood force calculation in molecular dynamics application. We have a 3D array of `Cell`

objects to hold the atom coordinates, and a 6D array of CellPair objects to perform pairwise force calculation between neighboring cells. In this case, not all elements in the 6D array of CellPair are necessary in the program. Only those which represent two immediately neighboring cells are needed for the force calculation. In this case, Charisma provides flexibility of declaring a sparse object array, with a `sparse` keyword following the object array declaration, as follows.

```
class CellPair : ChareArray6D;
obj cellpairs : CellPair[N,N,N,N,N,N], sparse;
```

Then the programmer is expected to supply a sequential function with the name `getIndex_ARRAYNAME` to generate a list of selected indices of the elements to create. As an example, the following function essentially tells the system to generate all the $N \times N \times N \times N \times N \times N$ elements for the 6D array.

```
void getIndex_cellpairs(CkVec<CkArrayIndex6D>& vec) {
    int i, j, k, l, m, n;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            for(k=0; k<N; k++)
                for(l=0; l<N; l++)
                    for(m=0; m<N; m++)
                        for(n=0; n<N; n++)
                            vec.push_back(CkArrayIndex6D(i, j, k, l, m, n));
}
```

17.9 Example: Jacobi 1D

Following is the content of the orchestration file `jacobi.or`.

```
program jacobi

class JacobiMain : MainChare;
class JacobiWorker : ChareArray1D;
obj workers : JacobiWorker[M];
param lb : double[N];
param rb : double[N];

begin
    for iter = 1 to MAX_ITER
        foreach i in workers
            (lb[i], rb[i]) <- workers[i].produceBorders();
            workers[i].compute(lb[i+1], rb[i-1]);
        end-foreach
    end-for
end
```

The class `JacobiMain` does not need any sequential code, so the only sequential code are in `JacobiWorker.h` and `JacobiWorker.C`. Note that `JacobiWorker.h` contains only the sequential portion of `JacobiWorker`'s declaration.

```
#define N 512
#define M 16

int currentArray;
double localData[2][M][N];
double localLB[N];
```

(continues on next page)

(continued from previous page)

```
double localRB[N];
int myLeft, myRight, myUpper, myLower;

void initialize();
void compute(double lghost[], double rghost[]);
void produceBorders(outport lb, outport rb);
double abs(double d);
```

Similarly, the sequential C code will be integrated into the generated C file. Below is part of the sequential C code taken from `JacobiWorker.C` to show how consumed parameters (`rghost` and `lghost` in `JacobiWorker::compute`) and produced parameters (`lb` and `rb` in `JacobiWorker::produceBorders`) are handled.

```
void JacobiWorker::compute(double rghost[], double lghost[]) {
    /* local computation for updating elements*/
}

void JacobiWorker::produceBorders(outport lb, outport rb) {
    produce(lb, localData[currentArray][myLeft], myLower-myUpper+1);
    produce(rb, localData[currentArray][myRight], myLower-myUpper+1);
}
```

The user compile these input files with the following command:

```
$ orhc jacobi.or
```

The compiler generates the parallel code for sending out messages, organizing flow of control, and then it looks for sequential code files for the classes declared, namely `JacobiMain` and `JacobiWorker`, and integrates them into the final output: `jacobi.h`, `jacobi.C` and `jacobi.ci`, which is a Charm++ program and can be built the way a Charm++ program is built.

Parallel Framework for Unstructured Meshes (ParFUM)

Contents

- *Parallel Framework for Unstructured Meshes (ParFUM)*
 - *Introduction*
 - * *Philosophy*
 - * *Terminology*
 - *Program Structure, Compilation and Execution*
 - * *Getting ParFUM*
 - * *Structure of a Typical ParFUM Program*
 - * *ParFUM Programs without init/driver*
 - * *Compilation*
 - * *Execution*
 - *ParFUM API Reference*
 - * *Utilities*
 - * *Basic Mesh Operations*
 - * *Mesh Entities*
 - * *Meshes*
 - * *Mesh Communication: Ghost Layers*
 - * *Older Mesh Operations*
 - * *Mesh Modification*
 - * *Topological Mesh Data*

- * *Mesh Adaptivity*
- * *Verifying correctness*
- * *Communication*
- * *Index Lists*
- * *Data Layout*
- * *IDXL Communication*
- * *Old Communication Routines*

18.1 Introduction

TERRY

18.1.1 Philosophy

TERRY

18.1.2 Terminology

TERRY

18.2 Program Structure, Compilation and Execution

18.2.1 Getting ParFUM

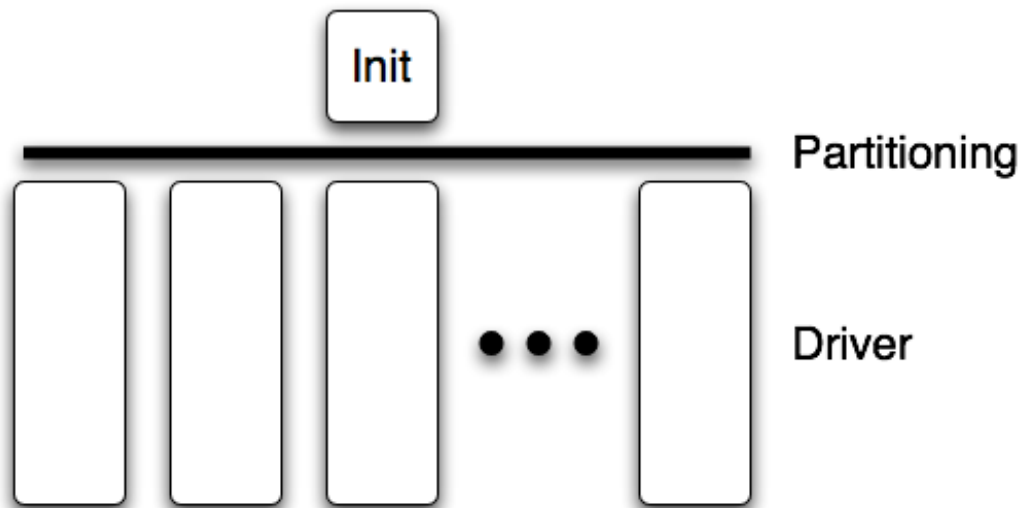
ParFUM is built on Charm++ so you must begin by downloading the latest source version of Charm++ from <http://charm.cs.illinois.edu/>. Build the source by running `./build` and answering the interactive prompts, or by manually specifying the configuration you want to the build script. Make sure to build the Charm++ libraries, not just the core system.

In a charm installation, see `charm/examples/ParFUM/` for example and test programs.

18.2.2 Structure of a Typical ParFUM Program

A typical ParFUM program consists of two functions: `init()` and `driver`. The `init()` function runs only on the first processor, and typically does specialized I/O and startup tasks. In most ParFUM programs `init()` is primarily used to read in a serial mesh. Once `init()` completes, ParFUM partitions the mesh and distributes it among all processors. Then `driver()` is called for every chunk on every processor and performs the main work of the program. This program structure is shown in Figure 52. In the language of the TCHARM manual, `init()` runs in the serial context and `driver()` runs in the parallel context.

In pseudocode, a simple ParFUM program would have the following structure:



52: A typical ParFUM program consists of an `init()` function running in serial context and a `driver()` function running in parallel context.

```

subroutine init
    read the serial mesh and configuration data
end subroutine
/* after init, the FEM framework partitions the mesh */
subroutine driver
    get local mesh chunk
    time loop
        FEM computations
        communicate boundary conditions
        more FEM computations
    end time loop
end subroutine

```

18.2.3 ParFUM Programs without init/driver

Although ParFUM provides the `init/driver` structure as a convenience to the programmer, you can write a ParFUM program without using `init` or `driver`. This is a more flexible approach, but it is more complicated than an `init/driver` program.

In pseudocode, a ParFUM program with a stand-alone main function might look like this:

```

main program
    MPI_Init
    FEM_Init(MPI_COMM_WORLD)
    if (I am master processor)
        read mesh
    partition mesh
    time loop
        FEM computations
        communicate boundary conditions
        more FEM computations

```

(continues on next page)

(continued from previous page)

```
end time loop
end main program
```

In this mode, the FEM framework does not set a default reading or writing mesh, and does no partitioning; you must use the FEM_Mesh routines to create and partition your mesh. See the AMPI manual for details on how to declare the main routine, or the file `main.C` in ParFUM for an example of how to write a stand-alone main routine. Compiling a ParFUM program without `init` or `driver` requires slightly different link flags than a typical ParFUM program, see the compilation section for details.

18.2.4 Compilation

To compile and link a ParFUM program, you must first have a working copy of Charm++ and the ParFUM libraries. The process for downloading and building this software is described in section 18.2.1.

To compile a FEM program, compile and link using `charmcc`, and pass the flag `-language ParFUM` to `charmcc` when linking. If your program uses its own `main` function rather than `init` and `driver`, pass `-language AMPI` instead.

18.2.5 Execution

At runtime, a Charm++/FEM framework program accepts the following options, in addition to all the usual Charm++ options described in the Charm++ “Installation and Usage Manual”.

- `+vp v`

Create v mesh chunks, or “virtual processors”. By default, the number of mesh chunks is equal to the number of physical processors (set with `+p p`).

- `-write`

Skip `driver()`. After running `init()` normally, the framework partitions the mesh, writes the mesh partitions to files, and exits. As usual, the `+vp v` option controls the number of mesh partitions.

This option is only used in programs with an `init` function.

- `-read`

Skip `init()`. The framework reads the partitioned input mesh from files and calls `driver()`. Together with `-write`, this option allows you to separate out the mesh preparation and partitioning phase from the actual parallel solution run.

This can be useful, for example, if `init()` requires more memory to hold the unpartitioned mesh than is available on one processor of the parallel machine. To avoid this limitation, you can run the program with `-write` on a machine with a lot of memory to prepare the input files, then copy the files and run with `-read` on a machine with a lot of processors.

`-read` can also be useful during debugging or performance tuning, by skipping the (potentially slow) mesh preparation phase. This option is only used in programs with a `driver` function.

- `+tcharm_trace fem`

Give a diagnostic printout on every call into the ParFUM framework. This can be useful for locating a sudden crash, or understanding how the program and framework interact. Because printing the diagnostics can slow a program down, use this option with care.

18.3 ParFUM API Reference

TERRY

18.3.1 Utilities

ISAAC

18.3.2 Basic Mesh Operations

TERRY

18.3.3 Mesh Entities

TERRY

Nodes

TERRY

Elements

TERRY

Sparse Elements

TERRY

Mesh Entity Operations

TERRY

Mesh Entity Queries

TERRY

Advanced Mesh Entity Operations

TERRY

18.3.4 Meshes

TERRY

Basic Mesh Operations

TERRY

Mesh Utilities

TERRY

Advanced Mesh Operations

TERRY

18.3.5 Mesh Communication: Ghost Layers

SAYANTAN

Ghost Numbering

SAYANTAN

Ghost Layer Creation

SAYANTAN

Symmetries and Ghosts: Geometric Layer

SAYANTAN

Advanced Symmetries and Ghosts: Lower Layer

SAYANTAN

18.3.6 Older Mesh Operations

SAYANTAN

Mesh Data Operations

SAYANTAN

Ghost Numbering

SAYANTAN

Backward Compatibility

SAYANTAN

Sparse Data

SAYANTAN

18.3.7 Mesh Modification

AARON

18.3.8 Topological Mesh Data

A ParFUM application can request that the ParFUM framework compute topological adjacencies. All ParFUM applications initially specify the mesh as a set of elements, each element defined by a fixed number of nodes. ParFUM can compute and maintain other sets of adjacencies such as which elements are adjacent to a given node, or which nodes are adjacent (they are both associated with a single element), or which elements share an edge/face with another element. Currently only a single element type is supported, and that element must be `FEM_ELEM+0`. To generate the structures storing the other types of adjacencies, each process in the ParFUM application should call the following subroutines:

`FEM_Add_elem2face_tuples(int mesh, 0, nodesPerFace, numFacesPerElement, faces);` specifies the topology of an element, specifically the configuration of its faces (if 3D) or edges (if 2D). Two elements are adjacent if they share a common face. The parameter `faces` is an integer array of length `nodesPerFace * numFacesPerElement`. The description is the same as used for determining ghost layers in section 18.3.5.

`FEM_Mesh_allocate_valid_attr(int mesh, int entity_type);`

`FEM_Mesh_create_node_elem_adjacency(int mesh);`

`FEM_Mesh_create_node_node_adjacency(int mesh);`

`FEM_Mesh_create_elem_elem_adjacency(int mesh);`

These subroutines can be called in `init` on a sequential mesh, or after partitioning in `driver`. The adjacencies will contain references to ghost elements if the subroutines were called in `driver` when ghosts are used. The indexes to ghosts are negative integers which can easily be converted to positive indices by using the function `FEM_To_ghost_index(id)`. The C header `ParFUM_internals.h` is required to be included by the ParFUM application to access the adjacencies. The functions to access the adjacencies are in sections 18.3.8, 18.3.8, and 18.3.8.

The internal data structures representing the adjacencies are maintained correctly when the adaptivity operations described in section 18.3.9 are used.

Accessing Element to Element Adjacencies

`void e2e_getAll(int e, int *neighbors);` places all of element `e`'s adjacent element ids in `neighbors`; assumes `neighbors` is already allocated to correct size

`int e2e_getNbr(int e, short idx);` returns the id of the `idx`-th adjacent element

Accessing Node to Element Adjacencies

`n2e_getLength(int n)` returns the number of elements adjacent to the given node `n`.

`n2e_getAll(int n, int *&adjelements, int &sz)` for node `n` place all the ids for adjacent elements into `adjelements`. You can ignore `sz` if you correctly determine the length beforehand.

Accessing Node to Node Adjacencies

`n2n_getLength(int n)` returns the number of nodes adjacent to the given node `n`.

`n2n_getAll(int n, int *&adjnodes, int &sz)` for node `n` place all the ids for adjacent nodes into `adjnodes`. You can ignore `sz` if you correctly determine the length beforehand.

18.3.9 Mesh Adaptivity

Initialization

If a ParFUM application wants to use parallel mesh adaptivity, the first task is to call the initialization routine from the *driver* function. This creates the node and element adjacency information that is essential for the adaptivity operations. It also initializes all the mesh adaptivity related internal objects in the framework.

```
void FEM_ADAPT_Init(int meshID)
```

Initializes the mesh defined by `meshID` for the mesh adaptivity operations.

Preparing the Mesh

For every element entity in the mesh, there is a desired size entry for each element. This entry is called `meshSizing`. This `meshSizing` entry contains a metric that determines element quality. The default metric is the average of the length of the three edges of an element. ParFUM provides various mechanisms to set this field. Some of the adaptive operations use these metrics to maintain quality. In addition, there is another metric which is computed for each element and maintained during mesh adaptivity. This metric is the ratio of the longest side to the shortest altitude, and this value is not allowed to go beyond a certain limit in order to maintain element quality.

```
void FEM_ADAPT_SetElementSizeField(int meshID, int elem, double size);
```

For the mesh specified by `meshID`, for the element `elem`, we set the desired size for each element to be `size`.

```
void FEM_ADAPT_SetElementSizeField(int meshID, double \*sizes);
```

For the mesh specified by `meshID`, for the element `elem`, we set the desired size for each element from the corresponding entry in the `sizes` array.

```
void FEM_ADAPT_SetReferenceMesh(int meshID);
```

For each element `int` in this mesh defined by `meshID` set its size to the average edge length of the corresponding element.

```
void FEM_ADAPT_GradateMesh(int meshID, double smoothness);
```

Resize mesh elements to avoid jumps in element size. That is, avoid discontinuities in the desired sizes for elements of a mesh by smoothing them out. Algorithm based on h-shock correction, described in Mesh Gradation Control, Borouchaki et al.

Modifying the Mesh

Once the elements in the mesh have been prepared by specifying their desired sizes, we are ready to use the actual adaptivity operations. Currently we provide Delaunay flip operations, edge bisect operations and edge coarsen operations, all of which are implemented in parallel. We provide several higher level functions which use these basic operations to generate a mesh with higher quality elements while achieving the desired sizing.

```
void FEM_ADAPT_Refine(int meshID, int qm, int method, double factor, double
\*sizes);
```

Perform refinements on the mesh specified by meshID. Tries to maintain/improve element quality by refining the mesh as specified by a quality measure qm. If method = 0, refine areas with size larger than factor down to factor. If method = 1, refine elements down to sizes specified in the sizes array. In this array each entry corresponds to the corresponding element. Negative entries in sizes array indicate no refinement.

```
void FEM_ADAPT_Coarsen(int meshID, int qm, int method, double factor, double
\*sizes);
```

Perform refinements on the mesh specified by meshID. Tries to maintain/improve element quality by coarsening the mesh as specified by a quality measure qm. If method = 0, coarsen areas with size smaller than factor down to factor. If method = 1, coarsen elements up to sizes specified in the sizes array. In this array each entry corresponds to the corresponding element. Negative entries in sizes array indicate no coarsening.

```
void FEM_ADAPT_AdaptMesh(int meshID, int qm, int method, double factor, double
\*sizes);
```

This function has the same set of arguments as required by the previous two operations, namely refine and coarsen. This function keeps using the above two functions until we have all elements in the mesh with as close to the desired quality. Apart from using the above two operations, it also performs a mesh repair operation which gets rid of some bad quality elements by Delaunay flip or coarsening as the geometry in the area demands.

```
int FEM_ADAPT_SimpleRefineMesh(int meshID, double targetA, double xmin, double
ymax, double xmax, double ymin);
```

A region is defined by (xmax, xmin, ymax, ymin) and the target area to be achieved for all elements in this region in the mesh specified by meshID is given by targetA. This function only performs a series of refinements on the elements in this region. If the area is larger, then no coarsening is done.

```
int FEM_ADAPT_SimpleCoarsenMesh(int meshID, double targetA, double xmin,
double ymin, double xmax, double ymax);
```

A region is defined by (xmax, xmin, ymax, ymin) and the target area to be achieved for all elements in this region in the mesh specified by meshID is given by targetA. This function only performs a series of coarsenings on the elements in this region. If the area is smaller, then no refinement is done.

18.3.10 Verifying correctness

We provide a checking function that can be used for debugging purposes to identify corrupted meshes or low quality elements.

```
void FEM_ADAPT_TestMesh(int meshID);
```

This provides a series of tests to determine the consistency of the mesh specified by meshID.

18.3.11 Communication

SAYANTAN

18.3.12 Index Lists

SAYANTAN

Index List Calls

SAYANTAN

Advanced Index List Calls

SAYANTAN

18.3.13 Data Layout

SAYANTAN

Layout Routines

SAYANTAN

Advanced Layout Routines

SAYANTAN

Layout Compatibility Routines

SAYANTAN

18.3.14 IDXL Communication

SAYANTAN

Communication Routines

SAYANTAN

Advanced Communication Routines

SAYANTAN

18.3.15 Old Communication Routines

SAYANTAN

Ghost Communication

SAYANTAN

Ghost List Exchange

SAYANTAN

CHAPTER 19

Charm++/Converse license

Non-Commercial Non-Exclusive License for Charm++ Software

The Software (as defined below) will only be licensed to You (as defined below) upon the condition that You accept all of the terms and conditions contained in this license agreement ("License"). Please read this License carefully. By downloading and using the Software, You accept the terms and conditions of this License.

1 Definitions.

- 1.1 "Illinois" means The Board of Trustees of the University of Illinois.
- 1.2 "Software" means the Charm++/Converse parallel programming software source code and object code.
- 1.3 "You" (or "Your") means an individual or legal entity exercising rights under, and complying with all of the terms of, this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with You. For purposes of this License, "control" means ownership, directly or indirectly, of more than fifty percent (50%) of the equity capital of the legal entity.

2 License Grant. Subject to Your compliance with the terms and conditions of this License, Illinois on behalf of Prof. Sanjay Kale's Parallel Programming Laboratory (PPL) group in the Department of Computer Science, hereby grants You a non-commercial, non-exclusive, non-transferable, royalty-free, restrictive license to download, use, and create derivative works of the Software for academic and research purposes only.

License Restrictions. Any uses of Software or Software derivative works beyond those granted in Section 2 above require a separate for a

(continues on next page)

(continued from previous page)

fee license. To negotiate a license with additional rights, You may contact Illinois at the Office of Technology Management at otm@illinois.edu or Charmworks, Inc. at info@hpccharm.com.

3 The following are some of the restricted use cases:

3.1 Integration of all or part of the Software or Software derivative works into a product for sale, lease or license by or on behalf of You to third parties; or

3.2 Distribution of all or part of the Software or Software derivative works to third parties for any reason, including for commercializing products sold or licensed by or on behalf of You, except for contributing changes to Software;

3.3 Use of Software or Software derivative works for internal commercial purposes.

3.4 For avoidance of doubt, You may at Your own expense, create and freely distribute Your works that interoperate with the Software, directing the users of such works to license and download the Software itself from Illinois' site at <http://charm.cs.illinois.edu/research/charm>.

4 Derivative works: You may at Your own expense, modify the Software to make derivative works as outlined in Section 2. Any derivative work should be clearly marked and renamed to notify users that it is a modified version and not the original Software distributed by Illinois. You agree to reproduce the copyright notice per Section 7 and other proprietary markings on any derivative work and to include in the documentation of such work the acknowledgement: "This software includes code developed by Parallel Programming Laboratory research group in the Department of Computer Science, at the University of Illinois at Urbana-Champaign."

5 Contributing derivative works: Software is being freely distributed as a research and teaching tool and as such, PPL research group encourages feedback and contributions from users of the code that might, at Illinois' sole discretion, be used or incorporated to make the basic operating framework of the Software a more stable, flexible, and/or useful product. If You wish to contribute Your code to become an internal portion of the Software:

5.1 You agree that such code may be distributed by Illinois under the terms of this License; and

5.2 You may be required to sign an "Agreement Regarding Contributory Code for Charm++ Software" (contact kale@illinois.edu), before Illinois can accept it.

6 Acknowledgment.

6.1 You shall own the results obtained by using the outputs of Software and Software derivative works. If You publish such results, You shall acknowledge the use of Software and its derivative works by the appropriate citation as follows:
"Charm++ software was developed by the Parallel Programming

(continues on next page)

(continued from previous page)

Laboratory research group in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

6.2 Any published work, which utilizes Software, shall include the following the references listed in the file named CITATION.cff in the distribution.

6.3 Electronic documents will include a direct link to the official Software page at <http://charm.cs.illinois.edu/research/charm>.

7 Copyright Notices. All copies of the Software or derivative works shall include the following copyright notice: "Copyright 2019 The Board of Trustees of the University of Illinois. All rights reserved."

8 Confidential Information. You acknowledge that the Software is proprietary to Illinois. You agree to protect the Software from unauthorized disclosure, use, or release and to treat the Software with at least the same level of care as You use to protect Your own proprietary software and/or confidential information, but in no event less than a reasonable standard of care. If You become aware of any unauthorized licensing, copying or use of the Software, You shall promptly notify Illinois in writing at otm@illinois.edu.

9 Limitation of Warranties. Limitation of Liability. Indemnification. THIS SOFTWARE IS PROVIDED "AS IS" AND ILLINOIS MAKES NO REPRESENTATIONS AND EXTENDS NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THAT THE USE OF THE SOFTWARE AND DERIVATIVE WORKS WILL NOT INFRINGE ANY PATENT, TRADEMARK, OR OTHER RIGHTS. YOU ASSUME THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE AND/OR DERIVATIVE WORKS. YOU AGREE THAT ILLINOIS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES WITH RESPECT TO ANY CLAIM BY YOU OR ANY THIRD PARTY ON ACCOUNT OF OR ARISING FROM THIS LICENSE OR USE OF THE SOFTWARE AND/OR DERIVATIVE WORKS. YOU SHALL INDEMNIFY AND HOLD HARMLESS ILLINOIS (INCLUDING ITS TRUSTEES, FELLOWS, OFFICERS, EMPLOYEES, STUDENTS, AND AGENTS) AGAINST ANY AND ALL CLAIMS, LOSSES, DAMAGES, AND/OR LIABILITY, AS WELL AS COSTS AND EXPENSES (INCLUDING LEGAL EXPENSES AND REASONABLE ATTORNEYS' FEES) ARISING OUT OF OR RELATED TO YOUR USE, OR INABILITY TO USE, THE SOFTWARE OR SOFTWARE DERIVATIVE WORKS, REGARDLESS OF THEORY OF LIABILITY, WHETHER FOR BREACH OR IN TORT (INCLUDING NEGLIGENCE).

10 Termination.

10.1 This License is effective until terminated, as provided herein, or until all intellectual property rights in the Software expire.

10.2 You may terminate this License at any time by destroying all copies of the Software and its derivative works.

10.3 This License, and the rights granted hereunder, will terminate automatically, and without any further notice from or action by Illinois, if You fail to comply with any obligation of this

(continues on next page)

(continued from previous page)

License.

- 10.4 Upon termination, You must immediately cease use of and destroy all copies of the Software and its derivative works and verify such destruction in writing.
- 10.5 The provisions set forth in Sections 5, 9, 11, 12, 13, 14, 15 shall survive termination or expiration of this License.
- 11 Governing Law. By using this Software, You agree to abide by the laws of the State of Illinois and all other applicable laws of the U.S, including, but not limited to, export control laws, copyright law and the terms of this License.
- 12 Severability. If any provision of this License is held to be invalid or unenforceable by a court of competent jurisdiction, such invalidity or unenforceability shall not in any way affect the validity or enforceability of the remaining provisions.
- 13 Assignment. You may not assign or otherwise transfer any of Your rights or obligations under this License, without the prior written consent of Illinois.
- 14 Entire License. This License represents the parties' entire agreement relating to the Software. Except as otherwise provided herein, no modification of this License is binding unless in writing and signed by an authorized representative of each party.
- 15 Waiver. The failure of either party to enforce any provision of this License shall not constitute a waiver of that right or future enforcement of that or any other provision.

Approved for legal form Illinois Counsel SJA 10/18

University Technology No. TF08023 Commercial Non-Exclusive Software License

Bibliography

- [PiP2018] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: techniques for practical address-space sharing. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18). ACM, New York, NY, USA, 131-143. DOI: <https://doi.org/10.1145/3208040.3208045>