Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

The Charm++ Parallel Programming System Manual

Version 6.4.0

University of Illinois
Charm++/Converse Parallel Programming System Software
Non-Exclusive, Non-Commercial Use License

Upon execution of this Agreement by the party identified below ("Licensee"), The Board of Trustees of the University of Illinois ("Illinois"), on behalf of The Parallel Programming Laboratory ("PPL") in the Department of Computer Science, will provide the Charm++/Converse Parallel Programming System software ("Charm++") in Binary Code and/or Source Code form ("Software") to Licensee, subject to the following terms and conditions. For purposes of this Agreement, Binary Code is the compiled code, which is ready to run on Licensee's computer. Source code consists of a set of files which contain the actual program commands that are compiled to form the Binary Code.

1. The Software is intellectual property owned by Illinois, and all right, title and interest, including copyright, remain with Illinois. Illinois grants, and Licensee hereby accepts, a restricted, non-exclusive, non-transferable license to use the Software for academic, research and internal business purposes only, e.g. not for commercial use (see Clause 7 below), without a fee.

2. Licensee may, at its own expense, create and freely distribute complimentary works that interoperate with the Software, directing others to the PPL server (`http://charm.cs.uiuc.edu`) to license and obtain the Software itself. Licensee may, at its own expense, modify the Software to make derivative works. Except as explicitly provided below, this License shall apply to any derivative work as it does to the original Software distributed by Illinois. 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. Licensee agrees to reproduce the copyright notice 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 the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

   Licensee may redistribute without restriction works with up to 1/2 of their non-comment source code derived from at most 1/10 of the non-comment source code developed by Illinois and contained in the Software, provided that the above directions for notice and acknowledgement are observed. Any other distribution of the Software or any derivative work requires a separate license with Illinois. Licensee may contact Illinois (`kale@cs.uiuc.edu`) to negotiate an appropriate license for such distribution.

3. Except as expressly set forth in this Agreement, 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 WILL NOT INFRINGE ANY PATENT, TRADEMARK, OR OTHER RIGHTS. LICENSEE ASSUMES THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS. LICENSEE AGREES THAT UNIVERSITY SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES WITH RESPECT TO ANY CLAIM BY LICENSEE OR ANY THIRD PARTY ON ACCOUNT OF OR ARISING FROM THIS AGREEMENT OR USE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS.

4. Licensee understands the Software is proprietary to Illinois. Licensee agrees to take all reasonable steps to insure that the Software is protected and secured from unauthorized disclosure, use, or release and will treat it with at least the same level of care as Licensee would use to protect and secure its own proprietary computer programs and/or information, but using no less than a reasonable standard of care. Licensee agrees to provide the Software only to any other person or entity who has registered with Illinois. If licensee is not registering as an individual but as an institution or corporation each member of the institution or corporation who has access to or uses Software must agree to and abide by the terms of this license. If Licensee becomes aware of any unauthorized licensing, copying or use of the Software, Licensee shall promptly notify Illinois in writing. Licensee expressly agrees to use the Software only in the manner and for the specific uses authorized in this Agreement.

5. By using or copying this Software, Licensee agrees to abide by the copyright law and all other applicable laws of the U.S. including, but not limited to, export control laws and the terms of this license. Illinois shall have the right to terminate this license immediately by written notice upon Licensee's breach of, or non-compliance with, any terms of the license. Licensee may be held legally responsible for any copyright infringement that is caused or encouraged by its failure to abide by the terms of this license. Upon termination, Licensee agrees to destroy all copies of the Software in its possession and to verify such destruction in writing.

6. The user agrees that any reports or published results obtained with the Software will acknowledge its use by the appropriate citation as follows:

   > "Charm++/Converse was developed by the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

   Any published work which utilizes Charm++ shall include the following reference:

   > "L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In 'Parallel Programming using C++' (Eds. Gregory V. Wilson and Paul Lu), pp 175-213, MIT Press, 1996."

   Any published work which utilizes Converse shall include the following reference:

   > "L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. Proceedings of the 10th International Parallel Processing Symposium, pp 212-217, April 1996."

   Electronic documents will include a direct link to the official Charm++ page at `http://charm.cs.uiuc.edu/`

7. Commercial use of the Software, or derivative works based thereon, REQUIRES A COMMERCIAL LICENSE. Should Licensee wish to make commercial use of the Software, Licensee will contact Illinois (kale@cs.uiuc.edu) to negotiate an appropriate license for such use. Commercial use includes:

   (a) integration of all or part of the Software into a product for sale, lease or license by or on behalf of Licensee to third parties, or

   (b) distribution of the Software to third parties that need it to commercialize product sold or licensed by or on behalf of Licensee.

8. Government Rights. Because substantial governmental funds have been used in the development of Charm++/Converse, any possession, use or sublicense of the Software by or to the United States government shall be subject to such required restrictions.

9. Charm++/Converse is being distributed as a research and teaching tool and as such, PPL encourages 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. Licensees who contribute their code to become an internal portion of the Software agree that such code may be distributed by Illinois under the terms of this License and may be required to sign an "Agreement Regarding Contributory Code for Charm++/Converse Software" before Illinois can accept it (contact `kale@cs.uiuc.edu` for a copy).

UNDERSTOOD AND AGREED.
Contact Information:
The best contact path for licensing issues is by e-mail to `kale@cs.uiuc.edu` or send correspondence to:

Prof. L. V. Kale
Dept. of Computer Science
University of Illinois
201 N. Goodwin Ave
Urbana, Illinois 61801 USA
FAX: (217) 333-3501

Contents

Chapter 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 BlueGene/L, BlueGene/P, BlueGene/Q, Cray XT, XE and XK series (including XK6 and XE6), a single workstation or a network of workstations (including x86 (running Linux, Windows, MacOS)), etc. The communication protocols and infrastructures supported by Charm++ are UDP, TCP, Myrinet, Infiniband, MPI, uGNI, and PAMI. Charm++ programs can run without changing the source on all these platforms. If built on MPI, Charm++ programs can also interoperate with pure MPI programs (§25). Please see the Charm++/Converse Installation and Usage Manual for details about installing, compiling and running Charm++ programs.

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.

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++: On each PE ("PE" stands for a "Processing Element". PEs are akin to processor cores; see section 1.4 for a precise description), 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, sometimes referred to as the Chare Kernel in the manual) 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 methods[1] are completely non-preemptive– Charm++ will not interrupt an executing method to start any other work, and all calls made are asynchronous.

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 3.3). 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 Kernel 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. 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 (Sec 5), which is the main notation we recommend you use. Alternatively, you may use threaded entry methods, in combination with *futures* and *sync* methods (See 12.1). 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.

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

The normal entry methods, being asynchronus, 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.

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 purprpose, 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 (Sec 2.9). 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 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].

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.

[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.

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 associoated 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 "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. Depending on the runtime command-line parmeters, a PE may be associated with a subset of cores or hardware threads. One or more PEs make up of a logical "node", the unit that one may partition a physical node into. In the implementation, a separate process exists for each logical node and all PEs within the logical node share the same memory address space. The Charm++ runtime system optimizes communication within a logical node by using shared memory.

For example, on a machine with 16-core nodes, where each core has two hardware threads, one may launch a Charm++ program with one or multiple (logical) nodes per physical node. One may choose 32 PEs per (logical) node, and one logical node per physical node. Alternatively, one can launch it with 12 PEs per logical node, and 1 logical node per physical node. One can also choose to partition the physical node, and launch it with 4 logical nodes per physical node (for example), and 4 PEs per 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 C. And utility functions to retrieve the information about those Charm++ logical machine entities in user programs can be refered in section 2.9.

Part I

Basic Charm++ Programming

Chapter 2

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.

2.1 .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 or describing 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.

2.2 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.

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

2.3 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. We adopt some conventions to ease this process. This is described in **??**.

2.4 Module Dependencies

A module may depend on the parallel entities declared in another module. It can express this dependency using the extern keyword. externed 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
    ...
};
```

The extern keyword places an include statement for the decl.h file of the externed module in the generated code of the current module. Hence, decl.h files generated from externed 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.

2.5 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 externed 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
    ...
};
```

2.6 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 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
    ...
};
```

2.7 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 **??**.

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.

2.8 Compiling Charm++ Programs

Charm++ provides a compiler-wrapper called charmc that handles all *ci*, C, C++ and fortran source files that are part of a user program. Users can invoke charmc 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. charmc and its functionality is described in B.

2.9 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 *message)

Causes the program to abort, printing the given error message. 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 time in seconds since the start of the program.

2.9.1 Information about Logical Machine Entities

As described in section 1.4, Charm++ recoginizes 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.

2.9.2 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.

Chapter 3

Basic Syntax

3.1 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 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 6 for more details on these routines.

Any user-defined types in the argument list must be declared 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.

3.2 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 processors[1]. 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 message[2].

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 3.1), or take a single argument that is a pointer to a Charm++ message (see section 10.1).

A chare's entry methods can be invoked via *proxies* (see section 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.

[1] Except when it is part of an array.

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

3.2.1 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.

3.2.2 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.

3.2.3 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
```

3.3 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 PE by the Charm++ runtime, and can be accessed in the same way as C++ "global" variables on any PE. Compound data structures containing pointers can be made available as read-only variables using read-only messages(see section 10.1) or read-only arrays(see section 4. 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 are pointers to message types. In this case, the message will be initialized on MessageType. Pointers are not allowed to be readonly variables unless they 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 Travelling Salesman Problem (TSP) in examples/charm++/hello, and GaussSeidel elimination in examples/charm++/gaussSeidel3D.

Chapter 4

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.

4.1 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);
    A(CkMigrateMessage *);

    void someEntry(parameters2);
};
```

Note that A must have a *migration constructor*, which is typically empty:

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

A::A(CkMigrateMessage *m) { /* the migration constructor */ }

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

See the section 6.3 on migratable array elements for more information on the migration constructor that takes CkMigrateMessage * as the argument.

4.2 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);}
```

The last declaration expects an array index of type CkArrayIndexFoo, which must be defined before including the `.decl.h` file (see section 13.3 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(){} ...};
```

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

4.3 Creating an Array

An array is created using the CProxy_Array::ckNew routine. This 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(parameters, dimX);
```

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

```
CProxy_ArrayB a2 = CProxy_ArrayB::ckNew(parameters, dimX, dimY);
CProxy_ArrayC a3 = CProxy_ArrayC::ckNew(parameters, dimX, dimY, dimZ);
```

For 4D, 5D, 6D and user-defined arrays, this functionality cannot be used. The array elements must be inserted individually as described in section 13.2.5.

During creation, the constructor is invoked on each array element. For more options when creating the array, see section 13.2.

4.4 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 dimentionality of the array:

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

Messages are not guarenteed 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.

[1] However, the element must eventually be created.

4.5 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.

4.6 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 4.6.1). 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:

```
// 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);
```

This 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.

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

Typically the client entry method of a reduction takes a single argument of type CkReductionMsg (see Section 16.1). 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 .cc 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 6). If you know that the reduction will yield a particular number of elements, say 3 `int`s, you can also specify a reduction target which takes 3 `int`s 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 16.1.

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

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

4.6.1 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 `int`, `float`, or `double` data as indicated by the suffix. The logical reductions (`and`, `or`) only work on 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_int, sum_float, sum_double– the result will be the sum of the given numbers.
3. CkReduction::product_int, product_float, product_double– the result will be the product of the given numbers.
4. CkReduction::max_int, max_float, max_double– the result will be the largest of the given numbers.
5. CkReduction::min_int, min_float, min_double– the result will be the smallest of the given numbers.
6. CkReduction::logical_and– the result will be the logical AND of the given integers. 0 is false, nonzero is true.
7. CkReduction::logical_or– the result will be the logical OR of the given integers.
8. CkReduction::bitvec_and– the result will be the bitvector AND of the given numbers (represented as integers).
9. CkReduction::bitvec_or– the result will be the bitvector OR of the given numbers (represented as integers).
10. 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.

11. CkReduction::concat– the result will be a byte-by-byte concatentation of all the contributed data. The contributed elements are not delimiter-separated.

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 below
  char data[];//The (dataSize-long) array of data
  CkReduction::setElement *next(void);
};
```

To extract the contribution of each array element from a reduction set, use the next routine repeatedly:

```
  //Inside a reduction handler--
  //  data is our reduced data from CkReduction_set
  CkReduction::setElement *cur=(CkReduction::setElement *)data;
  while (cur!=NULL)
  {
    ... //Use cur->dataSize and cur->data
    //Now advance to the next element's contribution
    cur=cur->next();
  }
```

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 see 6 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 heterogenous to be handled elegantly by the predefined types and you don't want to undertake multiple reductions, it may be best to define your own reduction type. See the next section (Section 16.2) for details.

4.7 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.

Chapter 5

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();
  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 {
      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. That expression will be compared for equality with the entry method's first argument, or with the reference number field of an explicit message (§ 10.1). 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
    . . .
  }
};
```

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++.

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 `forall` loop can be seen as an overlap with an indexed set of otherwise identical statements in the body. Its syntax is

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

The range from MIN to MAX is inclusive. Its use is demonstrated through distributed parallel matrix-matrix multiply shown in examples/charm++/matmul/matmul.ci

5.0.1 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.

5.1 Usage Notes

If you've added *Structured Dagger* code to your class, you must link in the code by:

- Adding "*className*_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).

- Adding a call to the pup routine "__sdag_pup(p);" from your pup routine. Forgetting this results in failure after migration.

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) {
       CBase_Foo::pup(p);
       __sdag_pup(p);
    }
    . . .
};
```

Chapter 6

Serialization Using the PUP Framework

The PUP framework is a generic way to describe the data in an object and to use that description for any task requiring 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. The name thus is a contraction of the words Pack and UnPack (PUP).

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 a foo to Charm++. This allows Charm++ to: marshall parameters of type foo across processors, translate foos across processor architectures, read and write foos to disk files, inspect and modify foo objects in the debugger, and checkpoint and restart calculations involving foos.

6.1 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.

6.1.1 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 17.2. 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];
}
```

6.1.2 PUP STL Container Objects

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

See examples/charm++/PUP/STLPUP

6.1.3 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 17.1 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

6.1.4 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 heterogenous architectures may become more common, so pupping as bytes is discouraged.

6.1.5 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 of builtin types like "int" and "double", or arrays of a type with the "PUPbytes" declaration, PUParray uses an even faster block transfer, with one virtual function call per array.

6.1.6 PUP structured dagger

Please note that if your object contains Structured Dagger code (see section 5) you must call the generated method __sdag_pup, after any superclass pup methods, to correctly pup the Structured Dagger state:

```
class bar : public barParent {
 public:
    bar_SDAG_CODE
    ...other methods...

    virtual void pup(PUP::er& p) {
      barParent::pup(p);
      __sdag_pup(p);
      ...pup other data here...
    }
};
```

6.1.7 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.

More specialized modes and PUP::ers are described in section 17.4.

6.2 PUP Usage Sequence

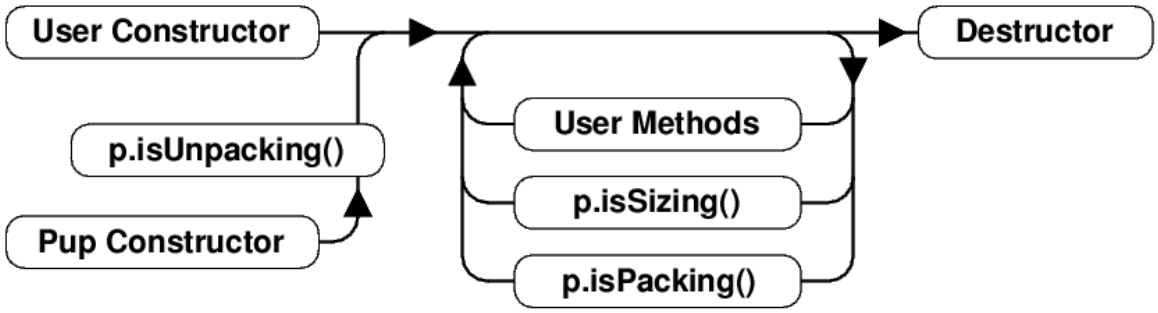

Figure 6.1: Method sequence of an object with a pup method.

Typical method invocation sequence of an object with a pup method is shown in Figure 6.1. 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 *"; 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.

6.3 Migratable Array Elements using PUP

Array objects can migrate from one PE to another. For example, the load balancer (see section 7.1) 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 6.1) and constraints wrt to serializing data, and use of Structured Dagger apply.

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

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

//In the .C file:
void A2::pup(PUP::er &p)
{
    CBase_A2::pup(p); //<- MUST call superclass's pup method
    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 7.1) 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 17.1).

6.4 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 6.1).

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 CkVec or 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

Chapter 7

Load Balancing

Load balancing in Charm++ is enabled by its ability to place, or migrate, chares or chare array elements. 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`

7.1 Measurement-based Object Migration Strategies

In Charm++, objects (except groups, nodegroups) 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, collects computation load and communication structure during execution and stores them into a load balancing database. 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. future can be to some extent predicted using the historical measurement data, allowing effective measurement-based load balancing without application-specific knowledge.

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

- Load balancing database provides the interface of almost all load balancing calls. On each processor, it stores the load balancing instrumented data and coordinates the load balancing manager and balancer. It is implemented as a Chare Group called LBDatabase.

- Load balancer or strategy takes the load balancing database and produces the new mapping of the objects. In Charm++, it is implemented as Chare Group inherited from BaseLB. Three kinds of schemes are implemented: (a) centralized load balancers, (b) fully distributed load balancers and (c) hierarchical load balancers.

7.2 Available Load Balancing Strategies

Load balancing can be performed in either a centralized, a fully distributed, or an hierarchical fashion.

In centralized approaches, the entire machine's load and communication structure are accumulated to a single point, typically processor 0, followed by a decision making process to determine the new distribution of Charm++ objects. Centralized load balancing requires synchronization which may incur an overhead and delay. However, due to the fact that the decision process has a high degree of the knowledge about the entire platform, it tends to be more accurate.

In distributed approaches, load data is only exchanged among neighboring processors. There is no global synchronization. However, they will not, in general, provide an immediate restoration for load balance - the process is iterated until the load balance can be achieved.

In hierarchical approaches, processors are divided into independent autonomous sets of processor groups and these groups are organized in hierarchies, thereby decentralizing the load balancing task. Different strategies can be used to balance the load on processors inside each processor group, and processors across groups in a hierarchical fashion.

Listed below are some of the available non-trivial centralized load balancers and their brief descriptions:

- **RandCentLB**: Randomly assigns objects to processors;

- **MetisLB**: Uses METIS™ to partitioning object communication graph.

- **GreedyLB**: Uses a greedy algorithm that always assigns the heaviest object to the least loaded processor.

- **GreedyCommLB**: Extends the greedy algorithm to take the communication graph into account.

- **TopoCentLB**: Extends the greedy algorithm to take processor topology into account.

- **RefineLB**: Moves objects away from the most overloaded processors to reach average, limits the number of objects migrated.

- **RefineSwapLB**: Moves objects away from the most overloaded processors to reach average. In case it cannot migrate an object from an overloaded processor to an underloaded processor, it swaps objects to reduce the load on the overloaded processor. This strategy limits the number of objects migrated.

- **RefineCommLB**: Same idea as in RefineLB, but takes communication into account.

- **RefineTopoLB**: Same idea as in RefineLB, but takes processor topology into account.

- **ComboCentLB**: A special load balancer that can be used to combine any number of centralized load balancers mentioned above.

Listed below are the distributed load balancers:

- **NeighborLB**: A neighborhood load balancer in which each processor tries to average out its load only among its neighbors.

- **WSLB**: A load balancer for workstation clusters, which can detect load changes on desktops (and other timeshared processors) and adjust load without interfering with other's use of the desktop.

An example of a hierarchical strategy can be found in:

- **HybridLB**: This calls GreedyLB at the lower level and RefineLB at the root.

Users can choose any load balancing strategy they think is appropriate for their application. The compiler and runtime options are described in section 7.6.

7.3 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 7.2 for a description of available load balancing strategies).

There are three 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 three methods are: **periodic load balancing mode**, **at sync mode** and **manual mode**.

In *periodic load balancing mode*, a user specifies only how often load balancing is to occur, using +LBPeriod runtime option to specify the time interval.

In *at sync mode*, the application invokes the load balancer explicitly at appropriate (generally at a pre-existing synchronization boundary) to trigger load balancing by inserting a function call (AtSync) in the application source code.

In the prior two load balancing modes, users do not need to worry about how to start load balancing. However, in one scenario, those automatic load balancers will fail to work - when array elements are created by dynamic insertion. This is because the above two load balancing modes require an application to have fixed the number of objects at the time of load balancing. The array manager needs to maintain a head count of local array elements for the local barrier. In this case, the application must use the *manual mode* to trigger load balancer.

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

1. **Periodical load balancing mode**: In the default setting, load balancing happens whenever the array elements are ready, with an interval of 1 second. It is desirable for the application to set a larger interval using +LBPeriod runtime option. For example "+LBPeriod 5.0" can be used to start load balancing roughly every 5 seconds. By default, array elements may be asked to migrate at any time, provided that they are not in the middle of executing an entry method. The array element's variable usesAtSync being CmiFalse attributes to this default behavior.

2. **At sync mode**: Using this method, elements can be migrated only at certain points in the execution when the application invokes AtSync(). In order to use the at sync mode, one should set usesAtSync to CmiTrue in the array element constructor. When an element is ready to migrate, call AtSync() [1]. When all local elements call AtSync, the load balancer is triggered. Once all migrations are completed, the load balancer calls the virtual function ArrayElement::ResumeFromSync() on each of the array elements. This function can be redefined in the application.

   Note that the minimum time for AtSync() load balancing to occur is controlled by the LBPeriod. Unusually high frequency load balancing (more frequent than 500ms) will perform better if this value is set via +LBPeriod or SetLBPeriod() to a number shorter than your load balancing interval.

   Note that *AtSync()* is not a blocking call, it just gives a hint to load balancing that it is time for load balancing. During the time between *AtSync* and *ResumeFromSync*, the object may be migrated. One can choose to let objects continue working with incoming messages, however keep in mind the object may suddenly show up in another processor and 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().

[1] AtSync() is a member function of class ArrayElement

The more commonly used approach is to force the object to be idle until load balancing finishes. The user places an AtSync call at the end of some iteration and when all 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 manual way of using the at sync mode results in a barrier at load balancing (see example here 7.8).

3. **Manual mode**: The load balancer can be programmed to be started manually. To switch to the manual mode, the application calls *TurnManualLBOn()* on every processor to prevent the load balancer from starting automatically. *TurnManualLBOn()* should be called as early as possible in the program. It could be called at the initialization part of the program, for example from a global variable constructor, or in an initcall(Section 9.1). It can also be called in the constructor of a static array and definitely before the *doneInserting* call for a dynamic array. It can be called multiple times on one processor, but only the last one takes effect.

   The function call *StartLB()* starts load balancing immediately. This call should be made at only one place on only one processor. This function is also not blocking, the object will continue to process messages and the load balancing when triggered happens in the background.

   TurnManualLBOff() turns off manual load balancing and switches back to the automatic Load balancing mode.

7.4 Migrating objects

Load balancers migrate objects automatically. For an array element to migrate, user can refer to Section 6.3 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 7.3).

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.

7.5 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(CmiBool 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 not[2].
- **LBSetPeriod(double s)**: this function can be called anywhere (even in Charm++ initcalls) to specify the load balancing period time in seconds. It tells load balancer not to start next load balancing in less than *s* seconds. This can be used to prevent load balancing from occurring too often in *automatic without sync mode*. Here is how to use it:

```
// if used in an array element
LBDatabase *lbdb = getLBDB();
lbdb->SetLBPeriod(5.0);

// if used outside of an array element
LBSetPeriod(5.0);
```

  Alternatively, one can specify +LBPeriod {seconds} at command line.

7.6 Compiler and runtime options to use load balancing module

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 different from activating it:

- link an LB module: is to link a Load Balancer module(library) at compile time. You can link against multiple LB libraries as candidates.
- activate an LB: is to actually ask the runtime to create an LB strategy and start it. You can only activate load balancers that have been linked at compile time.

Below are the descriptions about the compiler and runtime options:

1. **compile time options:**
   - *-module NeighborLB -module GreedyCommLB ...*
     links the modules NeighborLB, GreedyCommLB etc into an application, but these load balancers will remain inactive at execution time unless overridden by other runtime options.
   - *-module CommonLBs*
     links a special module CommonLBs which includes some commonly used Charm++ built-in load balancers. The commonly used load balancers include `BlockLB, CommLB, DummyLB, GreedyAgentLB, GreedyCommLB, GreedyLB, NeighborCommLB, NeighborLB, OrbLB, PhasebyArrayLB, RandCentLB, RecBipartLB, RefineLB, RefineCommLB, RotateLB, TreeMatchLB, RefineSwapLB, CommAwareRefineLB`.

[2]Currently not all load balancers recognize this setting though.

- *-balancer GreedyCommLB*
  links the load balancer GreedyCommLB and invokes it at runtime.
- *-balancer GreedyCommLB -balancer RefineLB*
  invokes GreedyCommLB at the first load balancing step and RefineLB in all subsequent load balancing steps.
- *-balancer ComboCentLB:GreedyLB,RefineLB*
  You can create a new combination load balancer made of multiple load balancers. In the above example, GreedyLB and RefineLB strategies are applied one after the other in each load balancing step.

The list of existing load balancers is given in Section 7.2. 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. One such load balancer, called MetisLB, requires the METIS library which is located at:

`charm/src/libs/ck-libs/parmetis/METISLib/.`

A pre-requisite for use of this library is to compile the METIS library by "make METIS" under charm/tmp after compiling Charm++.

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 GreedyCommLB*
     invokes GreedyCommLB
   - *+balancer GreedyCommLB +balancer RefineLB*
     invokes GreedyCommLB at the first load balancing step and RefineLB in all subsequent load balancing steps.
   - *+balancer ComboCentLB:GreedyLB,RefineLB*
     same as the example in the -balancer compile time option.

   Note: +balancer option works only if you have already linked the corresponding load balancers module at compile time. Giving +balancer with a wrong LB name will result in a runtime error. When you have used -balancer A as 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.

3. **Handling the case that no load balancer is activated by users**

   When no balancer is linked by users, but the program counts on a load balancer because it used *AtSync()* and expect *ResumeFromSync()* to be called to continue, a special load balancer called *NullLB* will be automatically created to run the program. This default load balancer calls *ResumeFromSync()* after *AtSync()*. It keeps a program from hanging after calling *AtSync()*. *NullLB* will be suppressed if another load balancer is created.

4. **Other useful runtime options**

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

   - *+LBDebug* {*verbose level*}
     {verbose level} can be any positive integer number. 0 is to turn off the verbose. This option asks load balancer to output load balancing information to stdout. The bigger the verbose level is, the more verbose the output is.

- *+LBPeriod {seconds}*
  {Seconds} can be any float number. This option sets the minimum period time in seconds between two consecutive load balancing steps. The default value is 1 second. That is to say that a load balancing step will not happen until 1 second after the last load balancing step.
- *+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.
- *+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.

7.7 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 **charmc** to link with the *foo* seed load balancer.

7.8 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;
    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 CmiTrue to make AtSync work
    usesAtSync=CmiTrue;
  }

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

  /* Must be written for migration to succeed */
  void pup(PUP::er &p){
    CBase_LBExample::pup(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"
```

Chapter 8

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 [1] 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).

8.1 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++ class [2].

8.1.1 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);
};
```

[1] Originally called *Branch Office Chare* or *Branched Chare*

[2] Older, deprecated syntax allows groups to inherit directly from the system-defined class Group

8.1.2 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.

```
// 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.

8.1.3 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.

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.

8.2 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.

8.2.1 NodeGroup Declaration

Node groups are defined in a similar way to groups. [3] 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.

8.2.2 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);
```

[3] 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 § 8.2.3 (below).

8.2.3 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 § 10.2.

8.2.4 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.

Chapter 9

Initializations at Program Startup

9.1 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 processor*.

```
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 processor 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.

9.2 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 § 8.1.2. 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.

Part II

Advanced Programming Techniques

Chapter 10

Optimizing Entry Method Invocation

10.1 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 is 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 an input message – consider an entry method invocation i_1, which receives as input the message buffer m_1. 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.

10.1.1 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 scenarious, 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 § 10.1.3.

10.1.2 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;
  MyPointerlessStruct var2;
  double arr3[10];

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

In particular, note the inclusion of the static array of `double`s, `arr3`, whose size is known at compile time to be that of ten `double`s. 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 (§ 10.3) 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.

10.1.3 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
  // 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 (alongwith 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
  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.

Immediate Messages

Immediate messages are special messages that skip the Charm scheduler, they can be executed in an "immediate" fashion even in the middle of a normal running entry method. They are supported only in nodegroup.

10.2 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, immediate, expedited, inline, local, python.

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 12.1.

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. 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 12.2.

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 `tests/charm++/pingpong`.

nokeep entry methods only take a message as the argument, and the memory buffer for this message will be managed by the Charm++ runtime rather than the user calls. This means that user has to guarantee that the message should not be buffered for a later usage or be freed in the user codes. Otherwise, a runtime error will be caused. Such entry methods entail runtime optimizations such as reusing the message memory. 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.

immediate entry methods are executed in an "immediate" fashion as they skip the message scheduling while other normal entry methods donot. Immediate entry methods should be only associcated with NodeGroup objects although it is not checked during compilation. 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. 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. An example "immediatering" can be found in `tests/charm++/megatest`.

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 immediately invoked if the message recipient happens to be on the same PE. These entry methods need to be re-entrant as they could be called multiple times recursively. If the recipient resides on a non-local PE, a regular message is sent, 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 reside on the same PE. Otherwise, the application will abort on a failure. If the local entry method uses parameter marshalling, instead of marshalling input parameters into a message, it will pass them direcly to the callee. This implies that the callee can modify the caller data if method parameters are passed by pointer or reference. Furthermore, input parameters do not require to be PUPable. Considering that these entry methods always execute immediately, they are allowed to have a non-void return value. Nevertheless, the return type of the method must be a pointer. An example can be found in `examples/charm++/hello/local`.

python entry methods are enabled to be called from python scripts as explained in chapter 21. Note that the object owning the method must also be declared with the keyword python. Refer to chapter 21 for more details.

reductiontarget entry methods may be used as the target of reductions, despite not taking CkReductionMsg as an argument. See section 4.6 for more references.

10.3 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 queuing strategy using one 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 message msg1 arrives at its destination, it will be pushed onto the end of the message queue as usual. However, when msg2 arrives, it will be pushed onto the *front* of the message queue. Similarly, the two parameter-marshalled calls to chare will be inserted at the end and beginning of the message queue, respectively.

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);
```

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 **??**. Although the size of the priority field is specified in bits, it is always padded to an integral number of `int`s. 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_IFIFO);
```

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 $.001001_{\text{binary}}$. 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.

Skipping 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 10.2 and the example in `tests/charm++/megatest/immediatering.ci`.

Chapter 11

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.

11.1 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(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.

2. 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.

3. CkCallback(CkCallback::ckExit) - When invoked, the callback will call CkExit(), ending the Charm++ program.

4. 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 myCB(CkIndex_myChare::myEntry(NULL), myChareProxy);
   ```

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.

One final type of callback, CkCallbackResumeThread(), can only be used from within threaded entry methods. This callback type is discussed in section 11.3.

11.2 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

11.3 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);
  ...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 examples/charm++/barnes-charm 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.

Chapter 12

Waiting for Completion

12.1 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.

12.2 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 must be in the form of a Charm++ message. 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.* § 12.1, 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.

12.3 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.

Charm++ provides all the necessary infrastructure to use futures by means of the following functions:

```
CkFuture CkCreateFuture(void)
void CkReleaseFuture(CkFuture fut)
int CkProbeFuture(CkFuture fut)
void *CkWaitFuture(CkFuture fut)
void  CkSendToFuture(CkFuture fut, void *msg)
```

To illustrate the use of all these functions, a Fibonacci example in Charm++ using futures in 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);
  }
}
```

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 Converse version of future functions can be found in the Converse manual.

12.4 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 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 `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(0), thisProxy),
                             CkCallback(CkIndex_chare2::finish_test(0), thisProxy), 0);
```

sets up completion detection for 10 producers. Once initialization is done, the callback associated with the `start_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.

12.5 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 (§ 11.1).

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.* § 12.1).

Chapter 13

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 optimisations.

13.1 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.

13.2 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.

13.2.1 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 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 signalled to the runtime by calling `opts.setAnytimeMigration(false)`[1]. If all array elements will be inserted by bulk creation or by `fooArray[x].insert()` calls, signal this by calling `opts.setStaticInsertion(true)` [2].

13.2.2 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);
  //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.

```
class BlockMap : public CkArrayMap
{
 public:
  BlockMap(void) {}
  BlockMap(CkMigrateMessage *m){}
  int registerArray(CkArrayIndex& numElements,CkArrayID aid) {
    return 0;
```

[1] At present, this optimizes broadcasts to not save old messages for immigrating chares.

[2] This can enable a slightly faster default mapping scheme.

```
  }
  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);
```

An example which contructs 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

13.2.3 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.

You can create your own set of elements by creating your own map object and overriding this virtual function of CkArrayMap:

```
  virtual void populateInitial(int arrayHdl,int numInitial,
void *msg,CkArrMgr *mgr)
```

In this call, arrayHdl is the value returned by registerArray, numInitial is the number of elements passed to CkArrayOptions, msg is the constructor message to pass, and mgr is the array to create.

populateInitial creates new array elements using the method void CkArrMgr::insertInitial(CkArrayIndex idx,void *ctorMsg). For example, to create one row of 2D array elements on each processor, you would write:

```
void xyElementMap::populateInitial(int arrayHdl,int numInitial,
void *msg,CkArrMgr *mgr)
{
  if (numInitial==0) return; //No initial elements requested

  //Create each local element
  int y=CkMyPe();
  for (int x=0;x<numInitial;x++) {
    mgr->insertInitial(CkArrayIndex2D(x,y),CkCopyMsg(&msg));
  }
  mgr->doneInserting();
  CkFreeMsg(msg);
}
```

Thus calling ckNew(10) on a 3-processor machine would result in 30 elements being created.

13.2.4 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 13.1), 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 advanatage 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

13.2.5 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. The insert 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=...)
   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 7.1)– 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 and load balancer sections for details.

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

13.2.6 Demand Creation

Demand Creation is a specialized form of dynamic insertion. Normally, invoking an entry method on a nonexistant 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

13.3 User-defined Array Indices

Charm++ array indices are arbitrary collections of integers. To define a new array index, you create an ordinary C++ class which inherits from CkArrayIndex and sets the "nInts" member to the length, in integers, of the array index.

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 <charm++.h>
class CkArrayIndexFoo:public CkArrayIndex {
    Foo f;
public:
    CkArrayIndexFoo(const Foo &in)
    {
        f=in;
        nInts=sizeof(f)/sizeof(int);
    }
    //Not required, but convenient: cast-to-foo operators
    operator Foo &() {return f;}
    operator const Foo &() const {return f;}
};
```

Note that Foo's 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
{ 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();
```

When you implement your array element class, as shown above you can inherit from CBase_ClassName, a class templated by the index type Foo. In the old syntax, you could also inherit directly from ArrayElementT. 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

Chapter 14

Sections: Subsets of a Chare Array

Charm++ supports defining and addressing subsets of the elements of a chare array. This entity is called a chare array section (section). These section elements are addressed via a section proxy. Charm++ also supports array sections which are a subset of array elements in multiple chare arrays of the same type 14.5. Multicast operations, a broadcast to all members of a section, are directly supported in array section proxy with an unoptimized direct-sending implementation. Section reduction is not directly supported by the section proxy. However, an optimized section multicast/reduction library called "CkMulticast" is provided as a separate library module, which can be plugged in as a delegation of a section proxy for performing section-based multicasts and reductions using optimized spanning trees.

14.1 Section Creation

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. In order to create an array section, the user needs to provide array indexes of all the array section members through either explicit enumeration, or an index range expression. You can create an array section proxy in your application by invoking ckNew() function of the CProxySection. For example, for a 3D array:

```
CkVec<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.push_back(CkArrayIndex3D(i, j, k));
CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, elems.getVec(), elems.size());
```

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 codes create a section proxy that contains array elements of [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.

```
CkArrayIndex *elems;    // add array indices
int numElems;
CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, elems, numElems);
```

14.2 Section Multicasts: via CkMulticast

Once you have the array section proxy, you can broadcast to all the section members, or send messages to one member using its offset index within the section, like these:

```
CProxySection_Hello proxy;
proxy.someEntry(...)          // section broadcast
proxy[0].someEntry(...)       // send to the first element in the section.
```

You can send the section proxy in a message to another processor, and still safely invoke the entry functions on the section proxy.

In the broadcast example above, for a section with k members, a total number of k messages will be sent, one to each member, which is inefficient when several members are on a same processor, in which case only one message needs to be sent to that processor and delivered to all section members on that processor locally. To support this optimization, a separate library called CkMulticast is provided as a target for delegation to an optimized implementation. This library also supports section based reduction.

Note: Use of the bulk array constructor (dimensions given in the CkNew or CkArrayOptions rather than individual insertion) will allow construction to race ahead of several other startup procedures, this creates some limitation on the construction delegation and use of array section proxies. For safety, array sections should be created in a post constructor entry method.

To use the library, you need to compile and install CkMulticast library and link your applications against the library using -module:

```
# compile and install the CkMulticast library, do this only once
# assuming a net-linux-x86_64 build
cd charm/net-linux-x86_64/tmp
make multicast

# link CkMulticast library using -module when compiling application
charmc  -o hello hello.o -module CkMulticast -language charm++
```

The CkMulticast library is implemented using delegation(Sec. 22). A special "CkMulticastMgr" Chare Group is created as a delegation for section multicast/reduction - all the messages sent by the section proxy will be passed to the local delegation branch.

To use the CkMulticast delegation, one needs to create the CkMulticastMgr Group first, and then setup the delegation relationship between the section proxy and CkMulticastMgr Group. One only needs to create one CkMulticastMgr Group globally. CkMulticastMgr group can serve all multicast/reduction delegations for different array sections in an application:

```
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
```

By default, CkMulticastMgr group builds a spanning tree for multicast/reduction with a factor of 2 (binary tree). One can specify a different factor when creating a CkMulticastMgr group. For example,

```
CkGroupID mCastGrpId = CProxy_CkMulticastMgr::ckNew(3);   // factor is 3
```

Note, to use CkMulticast library, all multicast messages must inherit from CkMcastBaseMsg, as the following. Note that CkMcastBaseMsg must come first, this is IMPORTANT for CkMulticast library to retrieve section information out of the message.

```
class HiMsg : public CkMcastBaseMsg, public CMessage_HiMsg
{
public:
  int *data;
};
```

Due to this restriction, you must define message explicitly for multicast entry functions and no parameter marshalling can be used for multicast with CkMulticast library.

14.3 Section Reductions

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 CkMulticastMgr 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;
  mcastGrp->contribute(sizeof(int), &data, CkReduction::sum_int, cookie);
}
```

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 sensive 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 reduction, to use section based reduction, 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);
mcastGrp->contribute(sizeof(int), &data, CkReduction::sum_int, cookie, cb);
```

If no member passes a callback to contribute, the reduction will use the default callback. You set the default callback for an array section using the setReductionClient call in the section root member. A **CkReductionMsg** message will be passed to this callback, which must delete the message when done.

```
CProxySection_Hello sectProxy;
CkMulticastMgr *mcastGrp = CProxy_CkMulticastMgr(mCastGrpId).ckLocalBranch();
mcastGrp->setReductionClient(sectProxy, new CkCallback(...));
```

As in an array reduction, users can use built-in reduction types(Section 4.6.1) or define his/her own reducer functions (Section 16.2).

14.4 Section Collectives when Migration Happens

Using multicast/reduction, you don't need to worry about array migrations. When migration happens, array element in the array section can still use the CkSectionInfo it stored previously for doing reduction. Reduction messages will be correctly delivered but may not be as efficient until a new multicast spanning tree is rebuilt internally in CkMulticastMgr library. When a new spanning tree is rebuilt, a 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 case when a multicast root migrates, the library must reconstruct the spanning tree to get optimal performance. One will get the following warning message if not doing so: "Warning: Multicast not optimized after multicast root migrated." In current implementation, 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
      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);
```

14.5 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 my all be of the same type.
- Each array must be enumerated by array ID
- The elements within each array must be enumerated explicitly
- No existing modules currently support delegation of cross section proxies. Therefore reductions are not currently supported.

Note: cross section logic also works for groups with analogous characteristics.
Given three arrays declared thusly:

```
CkArrayID *aidArr= new CkArrayID[3];
CProxy_multisectiontest_array1d *Aproxy= new CProxy_multisectiontest_array1d[3];
for(int i=0;i<3;i++)
  {
    Aproxy[i]=CProxy_multisectiontest_array1d::ckNew(masterproxy.ckGetGroupID(),ArraySize);
    aidArr[i]=Aproxy[i].ckGetArrayID();
  }
```

One can make a section including the lower half elements of all three arrays as follows:

```
int aboundary=ArraySize/2;
int afloor=aboundary;
int aceiling=ArraySize-1;
```

```
int asectionSize=aceiling-afloor+1;
// cross section lower half of each array
CkArrayIndex **aelems= new CkArrayIndex*[3];
aelems[0]= new CkArrayIndex[asectionSize];
aelems[1]= new CkArrayIndex[asectionSize];
aelems[2]= new CkArrayIndex[asectionSize];
int *naelems=new int[3];
for(int k=0;k<3;k++)
  {
    naelems[k]=asectionSize;
    for(int i=afloor,j=0;i<=aceiling;i++,j++)
      aelems[k][j]=CkArrayIndex1D(i);
  }
CProxySection_multisectiontest_array1d arrayLowProxy(3,aidArr,aelems,naelems);
```

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 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.

Chapter 15

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 accomodated 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);
  };

  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>;
  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);
};
```

Chapter 16

Collectives

16.1 Reduction Clients

After the data is reduced, it is passed to you via a callback object, as described in section 11. The message passed to the callback is of type CkReductionMsg. Unlike typed reductions briefed in Section 4.6, 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.

You may pass the client callback as an additional parameter to contribute. If different contribute calls pass different callbacks, some (unspecified, unreliable) callback will be chosen for use.

```
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);
```

If no member passes a callback to contribute, the reduction will use the *default* callback. Programmers can set the default callback for an array or group using the ckSetReductionClient proxy call on processor zero, or by passing the callback to `CkArrayOptions::setReductionClient()` before creating the array, as described in section 13.2.1. Again, a CkReductionMsg message will be passed to this callback, which must delete the message when done.

```
// Somewhere on processor zero:
myProxy.ckSetReductionClient(new CkCallback(...));
```

So, for the previous reduction on chare array `arr`:

```
CkCallback *cb = new CkCallback(CkIndex_main::reportIn(NULL),  mainProxy);
arr.ckSetReductionClient(cb);
```

and the actual entry point:

```
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++/RedExample for a complete example).

For backward compatibility, in the place of a general callback, you can specify a particular kind of C function using ckSetReductionClient or setReductionClient. This C function takes a user-defined parameter (passed to setReductionClient) and the actual reduction data, which it must not deallocate.

```
// Somewhere on processor zero (possibly in Main::Main, after creating 'myProxy'):
myProxy.setReductionClient(myClient,(void *)NULL);

// Code for the C function that serves as reduction client:
void myClient(void *param,int dataSize,void *data)
{
  double *forceSum=(double *)data;
  cout<<``First force sum is ``<<forceSum[0]<<endl;
  cout<<``Second force sum is ``<<forceSum[1]<<endl;
}
```

If the target of a reduction is an entry method defined by a *when* clause in SDAG(Section 5), 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, the contributors can pass an additional integer argument at the end of the `contribute()` call.

16.2 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 int`s 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));
    //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 from an initnode routine (see section 9.1 for details on the initnode mechanism). 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 4.6 can also be used for custom reductions. The target reduction client can be declared as in Section 4.6 but the reduction functions will be defined as explained above. Note that you cannot call CkReduction::addReducer from anywhere but an initnode routine.
(See examples/charm++/barnes-charm for a complete example).

Chapter 17

Serializing Complex Types

This section describes advanced functionality in the PUP framework. The first subsections describes 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.

17.1 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.

17.1.1 No allocation

The simplest case is when there is no dynamic allocation.

```
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() { }
};
```

17.1.2 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;}
};
```

17.1.3 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;}
};
```

17.1.4 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_;
      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;}
};
```

17.1.5 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.

17.1.6 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_;
      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;}
};
```

17.1.7 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.

17.2 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's constructor.
- A call to the macro PUPable_decl(className) in the class's declaration, in the header file. This adds a virtual method to your class to allow PUP::able to determine your class's type.
- 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.
- A working, virtual pup method. You can omit this if your class has no data that needs to be packed.

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 its subclass,

```
//In the .ci file:
   PUPable parent;
   PUPable child; //Could also have said ``PUPable parent, child;''

//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 ...
    }
};
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 ...
    }
};
```

With these declarations, then, 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);

// 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.
```

17.3 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.

17.4 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

17.5 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 disasterous 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.

17.6 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.

Chapter 18

Querying Network Topology

The following calls provide information about the machine upon which the parallel program is executed. A processing element (PE) refers to a single CPU, whereas a node refers to a single machine – a set of processing elements that share memory (i.e. an address space). PEs and nodes are ranked separately starting from zero, i.e., PEs are ranked in range 0 to *CmiNumPes()*, and nodes are ranked in range 0 to *CmiNumNodes()*.

Charm++ provides a unified abstraction for querying topology of IBM's BG/L, BG/P and BG/Q, and Cray's XT4, XT5 and XE6. Class TopoManager, which can be used by including TopoManager.h, contains following member functions:

TopoManager(): Default constructor.

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 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 rank of a processor, whose coordinates are known, on BG/P as well as on Cray XE6 using the following code:

```
TopoManager tmgr;
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.

Chapter 19

Checkpoint/Restart-Based Fault Tolerance

Charm++ offers a couple of 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 6), 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.

19.1 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 in *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`.

19.1.1 Checkpointing

The API to checkpoint the application is:

```
void CkStartCheckpoint(char* dirname,const CkCallback& cb);
```

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. 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 flexbility 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 and nodegroup objects (Section 8.1) are normally not meant to be migrated. In order to checkpoint them, however, the user has to write PUP routines for the groups 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 mainchare and declare them as `[migratable]` in the .ci file, just as in the case of Group and NodeGroup.

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.

19.1.2 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.

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 7) 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.

19.1.3 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.

19.2 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.

19.2.1 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 19.1.1. 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 call the following function to record a checkpoint:

```
void AMPI_MemCheckpoint()
```

19.2.2 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.

19.2.3 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
```

19.2.4 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++ net-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 19.2.6 . Charm++ runtime system can automatically detect failures and restart from checkpoint.

19.2.5 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 colum 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`.

19.2.6 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 communicataion. 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.

Part III

Expert-Level Functionality

Chapter 20

Tuning and Developing Load Balancers

20.1 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 correspoding 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.

20.2 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 learnt 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 learnt 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`.

20.3 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.

20.4 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 = CmiFalse;
```

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.

20.5 Writing a new load balancing strategy

Charm++ programmers can choose an existing load balancing strategy from Charm++'s built-in strategies(see 7.2) for the best performance based on the characteristics of their applications. However, they can also choose to write their own load balancing 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 20.1 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.

Incorporating this strategy into the Charm++ build framework is explained in the next section.

20.6 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.

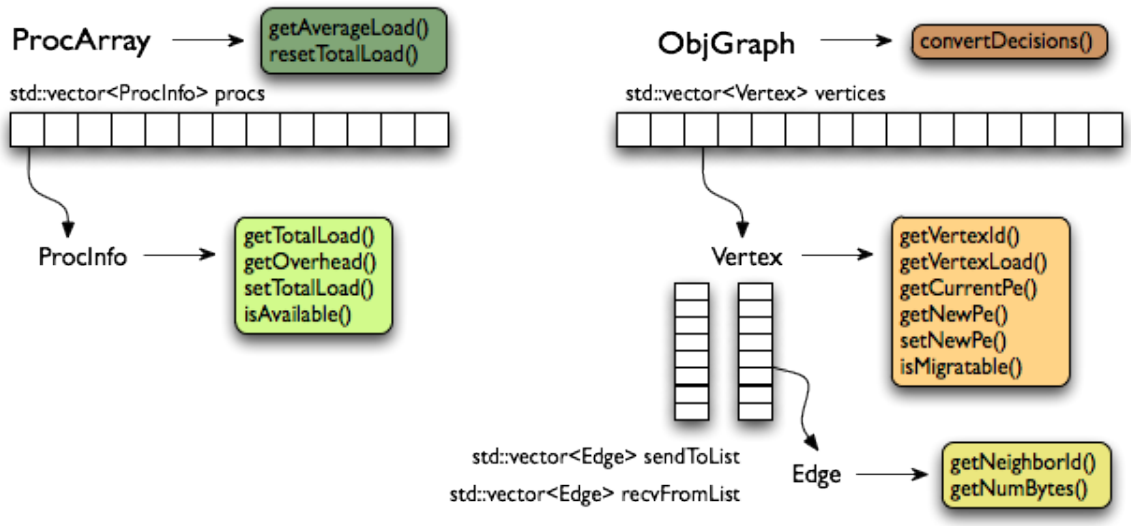

Figure 20.1: ProcArray and ObjGraph data structures to be used when writing a load balancing strategy

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.

20.7 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;
  CmiBool 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++ net-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*.

Chapter 21

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 Manual 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 21.8)
- **iteration** By implementing the iterator functions in the server (see 21.9), 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 21.11)

This documentation will describe the client API first, and then the server API.

21.1 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());
```

21.2 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 21.4). 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 21.4 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 PythonPrint 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.

21.3 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 21.2 for how to enable it, and 21.11 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 21.8 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 paramter, 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 21.8. Again, only two objects need to be passed, so extra parenthesis may be needed to create tuples from individual values.

21.4 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 21.9), 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 21.2), 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".

21.5 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();
```

21.6 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.

21.7 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’’);
```

21.8 Server **read** and **write** functions

As explained previously in subsection 21.3, 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`.

21.9 Server iterator functions

In order to use the iterative mode as explained in subsection 21.4, 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 21.10.

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.

21.10 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);

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".

21.11 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 11.2 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.

Chapter 22

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.

22.1 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.

22.2 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.

Part IV

Experimental Features

Chapter 23

Control Point Automatic Tuning

$\boxed{\beta}$

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

23.1 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 a 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 to cases. Sections 23.1.1 and 23.1.2 describe the additional changes to the program that should be made for each case.

23.1.1 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);
```

23.1.2 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
```

23.2 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.

23.3 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.

Chapter 24

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 mulithreading 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, just 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).

If there are no spare cores allocated, to avoid the resouce contention, a *unified runtime* is needed to supoort both the intra-node shared-memory multithreading parallelism and the inter-node distributed-memory message-passing parallelism. Additionally, considering a parallel application may just have a small fraction of its computation but critical that could be ported to shared-memory parallelism (the save on the critical computation may also reduce the communication cost, thus leading to more performance improvement), dedicating physical cores on every node to shared-memory multithreading runtime will cause a waste on computation power because those dedicated cores are not utilized at all during the most of application's execution time period. This case also indicates the necessity of a unified runtime so that both types of parallelism are supported.

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 re-uses Charm++ PEs to perform tasks spawned by the multithreading runtime. This library targets to be used in the Charm++ SMP mode.

The *CkLoop* library is built in $CHARM_DIR/$MACH_LAYER/tmp/libs/ck-libs/ckloop by executing "make" comamnd. To use it for user applications, one has to include "CkLoopAPI.h" in the source codes. The interface functions of this library are explained 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 initilization phase of the application. The argument "numThreads" is only used in the Charm++ non-SMP mode, specifying the number of threads to be created for the single-node shared-memory parallelism. It will be ignored in the SMP mode.

- void **CkLoop_Exit**(CProxy_FuncCkLoop ckLoop): This function is intended to be used in the non-SMP mode of Charm++ 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 a partial work on another thread */
  int paramNum, void * param, /* the input parameters for the above func */
  int numChunks, /* number of chunks to be partitioned */ int lowerRange, int upperRange, /* the loop-like parallelization happens in [lowerRange, upperRange] */

int sync=1, /* control the on/off of the implicit barrier after each parallelized loop */
void *redResult=NULL, REDUCTION_TYPE type=CKLOOP_NONE /* the reduction result, ONLY SUPPORT SINGLE VAR of TYPE int/float/double */
): The "HelperFn" is defined as "typedef void (*HelperFn)(int first,int last, void *result, int paramNum, void *param);" The "result" is the buffer for reduction result on a single simple-type variable.

Examples of using this library can be found in `examples/charm++/ckloop` and the widely-used molecule dynamics simulation application–NAMD[1]

[1] http://www.ks.uiuc.edu/Research/namd

Chapter 25

Charm-MPI Interoperation

Libraries written in Charm++ can also be used with pure MPI programs. Currently this functionality is supported only if Charm++ is built using MPI 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. We will be referring to this example program for ease of understanding.

25.1 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)

Here, *newComm* is the MPI communicator that Charm++ will use for the setup and communication. All the MPI ranks that belong to *newComm* should make this call. 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. As of now, a particular MPI rank can only be part of one unique Charm++ instance. Arguments *argc and argv* should contain the information required by Charm++ such as the load balancing strategy etc.

During the initialization the control is transferred from MPI to Charm++ RTS on the MPI ranks that made the call. Along with basic setup, Charm++ RTS also invokes the constructors of all mainchares during initialization. Once the intial 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 from MPI to Charm++ convenient (using pointers).

25.2 Writing Interoperable Charm++ Libraries

Minor modifications are required to make a Charm++ program interoperable with a pure MPI program:

- An interoperable Charm++ library should not contain a main module.
- *CkExit* should be used the same way *return* statement is used for returning back from a function call. It is advisable to make sure that only one chare invokes *CkExit*.
- Include *mpi-interoperate.h* - if not included, invoking *CkExit* will result in random output.

- 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. 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). One may 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.

25.3 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++.

25.4 Compilation

An interoperable Charm++ library can be compiled as usual using *charmc*. 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 *charmc*, 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 *charmc* 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.

Part V

Appendix

Appendix A

Installing Charm++

Charm++ can be installed either from the source code or using a precompiled binary package. 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.

A.1 Downloading Charm++

Charm++ can be downloaded using one of the following methods:

- From Charm++ website – The current stable version (source code and binaries) can be downloaded from our website at *http://charm.cs.illinois.edu/software*.
- From source archive – The latest development version of Charm++ can be downloaded from our source archive using *git clone git://charm.cs.illinois.edu/charm.git*.

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".

A.2 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 *net-linux-x86_64*, *bluegenep* 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.

In Table A.1, 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 appropiate choices for the build one wants to perform.

Net with 32 bit Linux	./build charm++ net-linux –with-production -j8
Multicore 64 bit Linux	./build charm++ multicore-linux64 –with-production -j8
Net with 64 bit Linux	./build charm++ net-linux-x86_64 –with-production -j8
Net with 64 bit Linux (intel compilers)	./build charm++ net-linux-x86_64 icc –with-production -j8
Net with 64 bit Linux (shared memory)	./build charm++ net-linux-x86_64 smp –with-production -j8
Net with 64 bit Linux (checkpoint restart based fault tolerance)	./build charm++ net-linux-x86_64 syncft –with-production -j8
MPI with 64 bit Linux	./build charm++ mpi-linux-x86_64 –with-production -j8
MPI with 64 bit Linux (shared memory)	./build charm++ mpi-linux-x86_64 smp –with-production -j8
MPI with 64 bit Linux (mpicxx wrappers)	./build charm++ mpi-linux-x86_64 mpicxx –with-production -j8
IBVERBS with 64 bit Linux	./build charm++ net-linux-x86_64 ibverbs –with-production -j8
Net with 32 bit Windows	./build charm++ net-win32 –with-production -j8
Net with 64 bit Windows	./build charm++ net-win64 –with-production -j8
MPI with 64 bit Windows	./build charm++ mpi-win64 –with-production -j8
Net with 64 bit Mac	./build charm++ net-darwin-x86_64 –with-production -j8
Blue Gene/L	./build charm++ bluegenel xlc –with-production -j8
Blue Gene/P	./build charm++ bluegenep xlc –with-production -j8
Blue Gene/Q	./build charm++ pami-bluegeneq xlc –with-production -j8
Cray XT3	./build charm++ mpi-crayxt3 –with-production -j8
Cray XT5	./build charm++ mpi-crayxt –with-production -j8
Cray XE6	./build charm++ gemini_gni-crayxe –with-production -j8

Table A.1: Build command for some common cases

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.

The main directories in a Charm++ installation are:

charm/bin Executables, such as charmc and charmrun, used by Charm++.

charm/doc Documentation for Charm++, such as this document. Distributed as LaTeX 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 libraries (.a) that comprise Charm++.

charm/pgms Example Charm++ programs.

charm/src Source code for Charm++ itself.

charm/tmp Directory where Charm++ is built.

charm/tools Visualization tools for Charm++ programs.

charm/tests Test Charm++ programs used by autobuild.

A.3 Security Issues

On most computers, Charm++ programs are simple binaries, and they pose no more security issues than any other program would. The only exception is the network version `net-*`, which has the following issues.

The network versions utilize many unix processes communicating with each other via UDP. Only a simple attempt is currently made to filter out unauthorized packets. Therefore, it is theoretically possible to mount a security attack by sending UDP packets to an executing Converse or Charm++ program's sockets.

The second security issue associated with networked programs is associated with the fact that we, the Charm++ developers, need evidence that our tools are being used. (Such evidence is useful in convincing funding agencies to continue to support our work.) To this end, we have inserted code in the network `charmrun` program (described later) to notify us that our software is being used. This notification is a single `UDP` packet sent by `charmrun` to `charm.cs.illinois.edu`. This data is put to one use only: it is gathered into tables recording the internet domains in which our software is being used, the number of individuals at each internet domain, and the frequency with which it is used.

We recognize that some users may have objections to our notification code. Therefore, we have provided a second copy of the `charmrun` program with the notification code removed. If you look within the `bin` directory, you will find these programs:

```
$ cd charm/bin
$ ls charmrun*
charmrun
charmrun-notify
charmrun-silent
```

The program `charmrun.silent` has the notification code removed. To permanently deactivate notification, you may use the version without the notification code:

```
$ cd charm/bin
$ cp charmrun.silent charmrun
```

The only versions of Charm++ that ever notify us are the network versions.

A.4 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`.

Appendix B

Compiling Charm++ Programs

The `charmc` program, located in "charm/bin", standardizes compiling and linking procedures among various machines and operating systems. "charmc" is a general-purpose tool for compiling and linking, not only restricted to Charm++ programs.

Charmc 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                            charmc -o pgm.o pgm.c
 * Compile C++                          charmc -o pgm.o pgm.C
 * Link                                 charmc -o pgm   obj1.o obj2.o obj3.o...
 * Compile + Link                       charmc -o pgm   src1.c src2.ci src3.C
 * Create Library                       charmc -o lib.a obj1.o obj2.o obj3.o...
 * Translate Charm++ Interface File     charmc file.ci
```

Charmc automatically figures out 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 charmc 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 charmc:

`-o` ***output-file*:** Output file name. Note: charmc 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 (eg, 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`.

`-D`*`symbol`*`[=`*`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 -O on the command line, it turns optimization back off. This is just a convenience for simple-minded 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 charmc 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, charmc prefers the less buggy of the two. This option causes charmc 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, charmc prefers the less buggy of the two, but not always. This option causes charmc to switch to the most reliable compiler, regardless of whether it produces slow code or not.

`-language {converse|charm++|ampi|fem|f90charm}`**:** When linking with charmc, one must specify the "language". This is just a way to help charmc 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 tolpogy 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.
- **verbose** Use a tracing set of memory routines. Every memory-related call results in a line printed to standard out. This version is useful for detecting memory leaks.
- **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 charmc to use the specified linker.

`-ld++` ***linker*****:** Use this option only when compiling programs that include C++ modules. Forces charmc 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.

Appendix C

Running Charm++ Programs

C.1 Launching Programs with charmrun

When compiling Charm++ programs, the charmc 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 C.2.1 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.

C.2 Command Line Options

A Charm++ program accepts the following command line options:

`+pN` Run the program with N processors. 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 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`.

C.2.1 Additional Network Options

The following ++ command line options are available in the network version:

`++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 rsh/ssh method.

This will pass `-n $P` to indicate how many processes to launch. An executable named something other than `mpiexec` can be used with the additional argument `++remote-shell` *runmpi*, with 'runmpi' replaced by the necessary name.

Use of this option can potentially provide a few benefits:

- Faster startup compared to the SSH/RSH 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`) and M(VA)PICH (`MPIRUN_RANK` and `MPIRUN_NPROCS` or `PMI_RANK` and `PMI_SIZE`).

`++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 `rsh` does not run the `.login` file.

`++maxrsh` Maximum number of `rsh`'s to run at a time.

`++nodelist` File containing list of nodes.

`++ppn` number of pes per node

`++help` print help messages

`++runscript` script to run node-program with

`++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

`++timeout` seconds to wait per host connection

`++p` number of processes to create

C.2.2 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, especailly when Charm++ applications has already implemented its 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), same runtime options can be used for all flavors of Charm++ versions including network and MPI versions, smp and non-smp versions.

`+setcpuaffinity` set cpu affinity automatically for processes (when Charm++ is based on non-smp versions) or threads (when smp)

`+excludecore <core #>` does 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.

`+pemap L[-U[:S[.R]]][,...]` Bind the execution threads to the sequence of cores described by the arguments using the operating system's CPU affinity functions.

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.

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`.

`+commap p[,q,...]` Bind communication threads to the listed cores, one per process.

C.2.3 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

C.3 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 "rsh" to spawn node processes on the remote hosts. The `shell` qualifier can be used to override it with say, "ssh". 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.

The user is required to set up remote login permissions on all nodes using the ".rhosts" file in the home directory if "rsh" is used for remote login into the hosts. If "ssh" is used, the user will have to setup password-less login to remote hosts using RSA authentication based on a key-pair and adding public keys to ".ssh/authorized_keys" file. See "ssh" documentation for more information.

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.

Appendix D

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
- Entry method attributes
  - stacksize
  - threaded
  - migratable
  - createhere
  - createhome
  - sync
  - iget
  - exclusive
  - immediate
  - expedited
  - inline
  - local
  - nokeep
  - notrace
  - python
  - accel
  - readwrite
  - writeonly
  - accelblock
  - memcritical
  - reductiontarget
- Basic C++ types
  - int
  - short
  - long
  - char
  - float
  - double
  - unsigned
  - void
  - const
- SDAG constructs
  - atomic
  - serial
  - forward

- when
- while
- for
- forall
- if
- else
- overlap
- connect
- publishes

Appendix E

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 E.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 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 E.1)

E.1 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 refereed 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.

E.1.1 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.
- `+binary-trace`: generate projections log in binary form.
- `+gz-trace`: generate gzip (if available) compressed log files.
- `+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`

E.1.2 Tracemode `summary`

Compile 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 an additional 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.

E.1.3 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. `0-31,32-999966:1000,999967-99999` to record every PE on the first 32, only every thousanth for the middle range, and the last 32 for a million processor run).

E.1.4 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 this`+traceWarn` 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.

E.2 Controlling Tracing from Within the Program

E.2.1 Selective Tracing

Charm++ allows user 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 E.1.3) if some period of time in middle of the applicationś 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.

E.2.2 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 `<br>`, `<b>`, `<i>`, `<font color=#ff00ff>`, 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 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 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.

Appendix F

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++.

Appendix G

Acknowledgements

- Aaron Becker
- Abhinav Bhatele
- Abhishek Gupta
- Akhil Langer
- Amit Sharma
- Anshu Arya
- Artem Shvorin
- Arun Singla
- Attila Gursoy
- Chao Huang
- Chao Mei
- Chee Wai Lee
- David Kunzman
- Dmitriy Ofman
- Edgar Solomonik
- Ehsan Totoni
- Emmanuel Jeannot
- Eric Bohm
- Eric Shook
- Esteban Meneses
- Esteban Pauli
- Filippo Gioachin
- Gengbin Zheng
- Greg Koenig

- Gunavardhan Kakulapati
- Hari Govind
- Harshitha Menon
- Isaac Dooley
- Jayant DeSouza
- Jeffrey Wright
- Jim Phillips
- Jonathan Booth
- Jonathan Lifflander
- Joshua Unger
- Josh Yelon
- Laxmikant Kale
- Lixia Shi
- Lukasz Wesolowski
- Mani Srinivas Potnuru
- Milind Bhandarkar
- Minas Charalambides
- Narain Jagathesan
- Neelam Saboo
- Nihit Desai
- Nikhil Jain
- Nilesh Choudhury
- Orion Lawlor
- Osman Sarood
- Parthasarathy Ramachandran
- Phil Miller
- Pritish Jetley
- Puneet Narula
- Rahul Joshi
- Ralf Gunter
- Ramkumar Vadali
- Ramprasad Venkataraman
- Rashmi Jyothi

- Robert Blake
- Robert Brunner
- Rui Liu
- Ryan Mokos
- Sameer Kumar
- Sameer Paranjpye
- Sanjeev Krishnan
- Sayantan Chakravorty
- Sindhura Bandhakavi
- Tarun Agarwal
- Terry L. Wilmarth
- Theckla Louchios
- Tim Hinrichs
- Timothy Knauff
- Vikas Mehta
- Viraj Paropkari
- Xiang Ni
- Yanhua Sun
- Yan Shi
- Yogesh Mehta
- Zheng Shao