Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

The CHARM++ Programming Language Manual

The Charm software was developed as a group effort. The earliest prototype, Chare Kernel(1.0), was developed by Wennie Shu and Kevin Nomura working with Laxmikant Kale. The second prototype, Chare Kernel(2.0), a complete re-write with major design changes, was developed by a team consisting of Wayne Fenton, Balkrishna Ramkumar, Vikram Saletore, Amitabh B. Sinha and Laxmikant Kale. The translator for Chare Kernel(2.0) was written by Manish Gupta. Charm(3.0), with significant design changes, was developed by a team consisting of Attila Gursoy, Balkrishna Ramkumar, Amitabh B. Sinha and Laxmikant Kale, with a new translator written by Nimish Shah. The CHARM++ implementation was done by Sanjeev Krishnan. Charm(4.0) included CHARM++ and was released in fall 1993. Charm(4.5) was developed by Attila Gursoy, Sanjeev Krishnan, Milind Bhandarkar, Joshua Yelon, Narain Jagathesan and Laxmikant Kale. Charm(4.8), developed by the same team included Converse, a parallel runtime system that allows interoperability among modules written using different paradigms within a single application. CHARM++ runtime system was re-targetted at Converse. Syntactic extensions in CHARM++ were dropped, and a simple interface translator was developed (by Sanjeev Krishnan and Jay DeSouza) that, along with the CHARM++ runtime, became the CHARM++ language. The current version (5.4R1) includes a complete rewrite of the CHARM++ runtime system (using C++) and the interface translator (done by Milind Bhandarkar). It also includes several new features such as Chare Arrays (developed by Robert Brunner and Orion Lawlor), and various libraries (written by Terry Wilmarth, Gengbin Zheng, Laxmikant Kale, Zehra Sura, Milind Bhandarkar, Robert Brunner, and Krishnan Varadarajan.) A coordination language "Structured Dagger" has been implemented on top of CHARM++ (Milind Bhandarkar), and included in this version. Several features have also been added to Converse. Dynamic seed-based load balancing has been implemented (Terry Wilmarth and Joshua Yelon), a client-server interface for Converse programs, and debugging support has been added (Parthasarathy Ramachandran, Jeff Wright, and Milind Bhandarkar). Converse has been ported to new platforms including ASCI Red (Joshua Yelon), Cray T3E (Robert Brunner), and SGI Origin2000 (Milind Bhandarkar). The test suite for CHARM++ was developed by Michael Lang, Jackie Wang, and Fang Hu. Projections, the performance visualization and analysis tool, was redesigned and rewritten using Java by Michael Denardo. Orion Lawlor, Gengbin Zheng, and Milind Bhandarkar have been responsible for the changes to the system since the last release.

Version 5.8 (Release 1)

Contents

1 Introduction

This manual describes Charm++, an object oriented portable parallel programming language based on C++. Its program structure, execution model, interface language constructs and runtime system calls are described here[1].

Charm++ has continuously evolved since the OOPSLA 1993 paper. The earlier versions modified the C++ syntax to support Charm++ primitives, and contained a full-fledged Charm++ translator that parsed the Charm++ syntactic extensions as well as the C++ syntax to produce a C++ program, which was later compiled using a C++ compiler. The current version does not augment the C++ syntax, and does not use a Charm++ translator as in previous versions. Instead, the older constructs are converted to calls into the runtime library, several new constructs are added, and minimal language constructs are used to describe the interfaces.

Charm++ is an explicitly parallel language based on C++ with a runtime library for supporting parallel computation called the Charm kernel. It provides a clear separation between sequential and parallel objects. The execution model of Charm++ is message driven, thus helping one write programs that are latency-tolerant. Charm++ supports dynamic load balancing while creating new work as well as periodically, based on object migration. Several dynamic load balancing strategies are provided. Charm++ supports both irregular as well as regular, data-parallel applications. It is based on the Converse interoperable runtime system for parallel programming.

Currently the parallel platforms supported by Charm++ are the PSC Lemieux, IBM SP, SGI Origin2000, Cray X1, Cray T3E, Intel Paragon, a single workstation or a network of workstations from Sun Microsystems (Solaris), IBM RS-6000 (AIX) SGI (IRIX 5.3 or 6.4), HP (HP-UX), Intel x86 (Linux, Windows 98/2000/XP) and Intel IA64. The communication protocols and infrastructures supported by Charm++ are UDP, TCP, Myrinet, Quadrics Elan, Shmem, MPI and VMI. Charm++ programs can run without changing the source on all these platforms. Please see the Charm++/Converse Installation and Usage Manual for details about installing, compiling and running Charm++ programs.

1.1 Overview

Charm++ is an object oriented parallel language. What sets Charm++ apart from traditional programming models such as message passing and shared variable programming is that the execution model of Charm++ is message-driven. Therefore, computations in Charm++ are triggered based on arrival of associated messages. These computations in turn can fire off more messages to other (possibly remote) processors that trigger more computations on those processors.

At the heart of any Charm++ program is a scheduler that repetitively chooses a message from the available pool of messages, and executes the computations associated with that message.

The programmer-visible entities in a Charm++ program are:

- Concurrent Objects : called *chares*[2]
- Communication Objects : Messages
- Readonly data

Charm++ starts a program by creating a single instance of each *mainchare* on processor 0, and invokes constructor methods of these chares. Typically, these chares then creates a number of other chares, possibly on other processors, which can simultaneously work to solve the problem at hand.

Each chare contains a number of *entry methods*, which are methods that can be invoked from remote processors. The Charm++ runtime system needs to be explicitly told about these methods, via an *interface* in a separate file. The syntax of this interface specification file is described in the later sections.

[1]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.

[2] Chare (pronounced **chär**, ä as in c**a**rt) is Old English for chore.

Charm++ provides system calls to asynchronously create remote chares and to asynchronously invoke entry methods on remote chares by sending messages to those chares. This asynchronous message passing is the basic interprocess communication mechanism in Charm++. However, Charm++ also permits wide variations on this mechanism to make it easy for the programmer to write programs that adapt to the dynamic runtime environment. These possible variations include prioritization (associating priorities with method invocations), conditional message packing and unpacking (for reducing messaging overhead), quiescence detection (for detecting completion of some phase of the program), and dynamic load balancing (during remote object creation). In addition, several libraries are built on top of Charm++ that can simplify otherwise arduous parallel programming tasks.

The following sections provide detailed information about various features of Charm++ programming system.

1.2 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 formally named the Charm Kernel, and was rewritten in C++. Several new features were added. The interface language underwent significant changes, and the macros that replaced the syntactic extensions in original Charm++, were replaced by natural C++ constructs. Late 1999, and early 2000 reflected several additions to Charm++, when a load balancing framework and migratable objects were added to Charm++.

2 Charm++ Overview

We think that Charm++ is easy to use if you are familiar with object-based programming. (But of course that is our opinion, if your opinion differs, you are encouraged to let us know the reasons, and features that you would like to see in Charm++.) Object-based programming is built around the concept of "encapsulation" of data. As implemented in C++, data encapsulation is achieved by grouping together data and methods (also known as functions, subroutines, or procedures) inside of an object.

A class is a blueprint for an object. The encapsulated data is said to be "private" to the object, and only the methods of that class can manipulate that data. A method that has the same name as the class is a "blessed" method, called a "Constructor" for that class. A constructor method is typically responsible for initializing the encapsulated data of an object. Each method, including the constructor can optionally be supplied data in the form of parameters (or arguments). In C++, one can create objects with the `new` operator that returns a pointer to the object. This pointer can be used to refer to the object, and call methods on that object.

Charm++ is built on top of C++, and also based on "encapsulation". Similar to C++, Charm++ entities can contain private data, and public methods. The major difference is that these methods can be invoked from remote processors asynchronously. Asynchronous method invocation means that the caller does not wait for the method to be actually executed and does not wait for the method's return value. Therefore, Charm++ methods (called entry methods) do not have a return value[3]. Since the actual Charm++ object on which the method is being invoked may be on a remote processor[4], the C++ way of referring to an object, via a pointer, is not valid in Charm++. Instead, we refer to a remote chare via a "proxy", as explained below.

2.1 Proxies and Handles

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.[5] 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 forwards this method invocation to the actual remote object. All entities that are created and manipulated remotely in Charm++ have such proxies. 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.

You can have several proxies that all refer to the same object.

Historically, handles (which are basically globally unique identifiers) were used to uniquely identify Charm++ objects. Unlike pointers, they are valid on all processors and so could be sent as parameters in messages. They are still available, but now proxies also have the same feature.

Handles (like CkChareID, CkArrayID, etc.) and proxies (like CProxy_foo) are just bytes and can be sent in messages, pup'd, and parameter marshalled. This is now true of almost all objects in Charm++: the only exceptions being entire Chares (Array Elements, etc.) and, paradoxically, messages themselves.

2.2 Charm++ Execution Model

A Charm++ program consists of a number of Charm++ objects distributed across the available number of processors. Thus, the basic unit of parallel computation in Charm++ programs is the *chare*, a Charm++ object that can be created on any available processor and can be accessed from remote processors. A chare is similar to a process, an actor, an ADA task, etc. Chares are created dynamically, and many chares may be active simultaneously. Chares send *messages* to one another to invoke methods asynchronously. Conceptually, the system maintains a "work-pool" consisting of seeds for new chares, and messages for existing chares. The runtime system (called *Charm Kernel*) may pick multiple items, non-deterministically, from this pool and execute them.

Methods of a chare that can be remotely invoked are called *entry* methods. Entry methods may take marshalled 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[6] are completely non-preemptive– Charm++ will never 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 Kernel will then place the remote chare on a least loaded processor. Thus one can imagine chare creation as generating only a seed for the new chare, which may *take root* on the most *fertile* processor. Charm Kernel identifies a chare by a *ChareID*. Since user code

[3]Asynchronous remote method invocation is the core of Charm++. However, to simplify programming, Charm++ makes use of the interoperable nature of its runtime system, and combines seamlessly with user-level threads to also support synchronous method execution, albeit with a slight overhead of thread creation and scheduling.

[4]With its own, different address space

[5]The proxy class is generated by the "interface translator" based on a description of the entry methods

[6]"Threaded" or "synchronous" methods are different.

does not need to name a chares' processor, chares can potentially migrate from one processor to another. (This behavior is used by the dynamic load-balancing framework for chare containers, such as arrays.)

Other CHARM++ objects are collections of chares. They are: *chare-arrays*, *chare-groups*, and *chare-nodegroups*, referred to as *arrays*, *groups*, and *nodegroups* throughout this manual. An array is a collection of 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, one per processor (SMP node), that is addressed using a unique system-wide name.

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.

The only method of communication between processors in CHARM++ is asynchronous entry method invocation on remote chares. For this purpose, Charm Kernel needs to know the types of chares in the user program, the methods that can be invoked on these chares from remote processors, the arguments these methods take as input etc. Therefore, when the program starts up, these user-defined entities need to be registered with Charm Kernel, which assigns a unique identifier to each of them. While invoking a method on a remote object, these identifiers need to be specified to Charm Kernel. Registration of user-defined entities, and maintaining these identifiers can be cumbersome. Fortunately, it is done automatically by the CHARM++ interface translator. The CHARM++ interface translator generates definitions for *proxy* objects. A proxy object acts as a *handle* to a remote chare. One invokes methods on a proxy object, which in turn carries out remote method invocation on the chare.

In addition, the CHARM++ interface translator provides ways to enhance the basic functionality of Charm Kernel using user-level threads and futures. These allow entry methods to be executed in separate user-level threads. These *threaded* entry methods may block waiting for data by making *synchronous* calls to remote object methods that return results in messages.

CHARM++ program execution is terminated by the CkExit call. Like the exit system call, CkExit never returns. The Charm Kernel 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.

2.3 Entities in Charm++ programs

This section describes various entities in a typical CHARM++ program.

2.3.1 Sequential Objects

A CHARM++ program typically consists mostly of ordinary sequential C++code and objects. Such entities are only accessible locally, are not known to the CHARM++ runtime system, and thus need not be mentioned in the module interface files.

CHARM++ does not affect the syntax or semantics of such C++ entities, except that changes to global variables (or static data members of a class) on one node will not be visible on other nodes. Global data changes must be explicitly sent between processors. For processor- and thread-private storage, refer to the "Global Variables" section of the Converse manual.

2.3.2 Messages

Messages supply data arguments to the asynchronous remote method invocation. These objects are treated differently from other objects in CHARM++ by the runtime system, and therefore they must be specified in the interface file of the module. With parameter marshalling, the system creates and handles the message completely internally. Other messages are instances of C++ classes that are subclassed from a special class that is generated by the CHARM++ interface translator. Another variation of communication objects is conditionally packed and unpacked. This variation should be used when one wants to send messages that contain pointers to the data rather than the actual data to other processors. This type of communication objects contains two static methods: pack, and unpack. The third variation of communication objects is called *varsize* messages. Varsize messages is an effective optimization on conditionally packed messages, and can be declared with special syntax in the interface file.

2.3.3 Chares

Chares are the most important entities in a CHARM++ program. These concurrent objects are different from sequential C++ objects in many ways. Syntactically, Chares are instances of C++ classes that are derived from a system-provided class called Chare. Also, in addition to the usual C++ private and public data and method members, they contain some public methods called *entry methods*. These entry methods do not return anything (they are void methods), and take at most one argument, which is a pointer to a message. Chares are *accessed* using a proxy (an object of a specialized class generated by the CHARM++ interface translator) or using a handle (a CkChareID structure defined in CHARM++), rather than a pointer as in C++. Semantically, they are different from C++ objects because they can be created asynchronously from remote processors, and their entry methods also could be invoked asynchronously from the remote processors. Since the constructor method is invoked from remote processor (while creating a chare), every chare should have its constructors as entry methods (with at most one message pointer parameter). These chares and their entry methods have to be specified in the interface file.

2.3.4 Chare Arrays

Chare arrays are collections of chares. However, unlike chare groups or nodegroups, arrays are not constrained by characteristics of the underlying parallel machine such as number of processors or nodes. Thus, chare arrays can have any number of *elements*. The array elements themselves are chares, and methods can be invoked on individual array elements as usual. Each element of an array has a globally unique index, and messages are addressed to that index.

Unlike other entities in CHARM++ the dynamic load balancing framework (LB Framework) treats array elements as objects that can be migrated across processors. Thus, the runtime system keeps track of computational load across the system, and also the time spent in execution of entry methods on array elements, and then employs one of several strategies to redistribute array elements across the available processors.

2.3.5 Chare Groups

Chare Groups[7] are a special type of concurrent objects. Each chare group is a collection of chares, with one representative (group member) on each processor. All the members of a chare group share a globally unique name (handle, defined by Charm kernel to be of type CkGroupID). An entire chare group could be addressed using this global handle, and an individual member of a chare group can be addressed using the global handle, and a processor number. Chare groups are instances of C++ classes subclassed from a system-provided class called Group. The Charm kernel has to be notified that these chares are semantically different, and therefore chare groups have a different declaration in the interface specification file.

2.3.6 Chare Nodegroups

Chare nodegroups are very similar to chare groups except that instead of having one group member on each processor, the nodegroup has one member on each shared memory multiprocessor node. Note that CHARM++ (and its underlying runtime system Converse) distinguish between processors and nodes. A node consists of one or more processors that share an address space. The last few years have seen emergence of fast SMP systems of small (2-4 processors) to large (32-64 processors) number of processors per node. A network of such SMP nodes is the most general model of parallel computers, making pure distributed and pure shared memory systems mere special cases. CHARM++ is built on top of this machine abstraction, and Chare nodegroups embody this abstraction in a higher level language construct. Semantically, methods invoked on a nodegroup member could be executed on any processor within that node. This fact can be

[7] These were called Branch Office Chares (BOC) in earlier versions of Charm.

utilized for supporting load balance across processors within a node. However, this also means that different processors within a node could be executing methods of the same nodegroup member simultaneously, thus leading to common problems associated with shared address space programming. However, Charm++ eases such problems by allowing the programmer to specify an entry method of a nodegroup to be *exclusive*, thus guaranteeing that no other *exclusive* method of that nodegroup member can execute simultaneously within the node.

3 The Charm++ Language

3.1 Modules

3.1.1 Structure of a Charm++ Program

A Charm++ program is structurally similar to a C++ program. Most of a Charm++ program *is* C++ code.[8] The main syntactic units in a Charm++ program are class definitions. A Charm++ program can be distributed across several source code files.

There are five disjoint categories of objects (classes) in Charm++:

- Sequential objects: as in C++
- Chares (concurrent objects)
- Chare Groups (a form of replicated objects)
- Chare Arrays (an indexed collection of chares)
- Messages (communication objects)

The user's code is written in C++ and interfaces with the Charm++ system as if it were a library containing base classes, functions, etc. A translator is used to generate the special code needed to handle Charm++ constructs. This translator generates C++ code that needs to be compiled with the user's code.

Interfaces to the Charm++ objects (such as messages, chares, readonly variables etc.) have to be declared in Charm++ interface files. Typically, such entities are grouped into *modules*. A Charm++ program may consists of multiple modules. One of these modules is declared to be a mainmodule. All the modules that are "reachable" from the mainmodule via the extern construct are included in a Charm++ program.

The Charm++ interface file has the suffix ".ci". The Charm++ interface translator parses this file and produces two files (with suffixes ".decl.h" and ".def.h", *for each module declared in the ".ci" file*), that contain declarations (interface) and definitions (implementation)of various translator-generated entities. If the name of a module is *MOD*, then the files produced by the Charm++ interface translator are named *MOD.decl.h* and *MOD.def.h*.[9] We recommend that the declarations header file be included at the top of the header file (*MOD.h*) for module *MOD*, and the definitions file be included at the bottom of the code for module (*MOD.C*).[10]

A simple Charm++ program is given below:

```
///////////////////////////////////////
// File: pgm.ci

mainmodule Hello {
  readonly CProxy_HelloMain mainProxy;
  mainchare HelloMain {
    entry HelloMain(); // implicit CkArgMsg * as argument
```

[8]**Constraint: The C++ code cannot, however, contain global or static variables.**

[9]Note that the interface file for module *MOD* need not be named *MOD.ci*. Indeed one ".ci" file may contain interface declarations for multiple modules, and the translator will produce one pair of declaration and definition files for each module.

[10]In the earlier version of interface translator, these files used to be suffixed with ".top.h" and ".bot.h" for this reason.

```
    entry void PrintDone(void);
  };
  group HelloGroup {
    entry HelloGroup(void);
  };
};

////////////////////////////////////////
// File: pgm.h
#include "Hello.decl.h" // Note: not pgm.decl.h

class HelloMain: public Chare {
  public:
    HelloMain(CkArgMsg *);
    void PrintDone(void);
  private:
    int count;
};

class HelloGroup: public Group {
  public:
    HelloGroup(void);
};

/////////////////////////////////////////
// File: pgm.C
#include "pgm.h"

CProxy_HelloMain mainProxy;

HelloMain::HelloMain(CkArgMsg *msg) {
  delete msg;
  count = 0;
  mainProxy=thishandle;
  CProxy_HelloGroup::ckNew(); // Create a new "HelloGroup"
}

void HelloMain::PrintDone(void) {
  count++;
  if (count == CkNumPes()) { // Wait for all group members to finish the printf
    CkExit();
  }
}

HelloGroup::HelloGroup(void) {
  ckout << "Hello World from processor " << CkMyPe() << endl;
  mainProxy.PrintDone();
}

#include "Hello.def.h" // Include the Charm++ object implementations

/////////////////////////////////////////
// File: Makefile
```

```
pgm: pgm.ci pgm.h pgm.C
      charmc -c pgm.ci
      charmc -c pgm.C
      charmc -o pgm pgm.o -language charm++
```

HelloMain is designated a mainchare. Thus the Charm Kernel starts execution of this program by creating an instance of *HelloMain* on processor 0. The HelloMain constructor creates a chare group *HelloGroup*, and stores a handle to itself and returns. The call to create the group returns immediately after directing Charm Kernel to perform the actual creation and invocation. Shortly after, the Charm Kernel will create an object of type *HelloGroup* on each processor, and call its constructor. The constructor will then print "Hello World..." and then call the *PrintDone* method of *HelloMain*. The *PrintDone* method calls CkExit after all group members have called it (i.e., they have finished printing "Hello World..."), and the CHARM++program exits.

3.1.2 Functions in the "decl.h" and "def.h" files

The `decl.h` file provides declarations for the proxy classes of the concurrent objects declared in the ".ci" file (from which the `decl.h` file is generated). So the *Hello.decl.h* file will have the declaration of the class CProxy_HelloMain. Similarly it will also have the declaration for the HelloGroup class.

This class will have functions to create new instances of the chares and groups, like the function ckNew. For *HelloGroup* this function creates an instance of the class *HelloGroup* on all the processors.

The proxy class also has functions corresponding to the entry methods defined in the ".ci" file. In the above program the method wait is declared in *CProxy_HelloMain* (proxy class for *HelloMain*).

The proxy class also provides static registration functions used by the CHARM++ runtime. The `def.h` file has a registration function (*__registerHello* in the above program) which calls all the registration functions corresponding to the readonly variables and entry methods declared in the module.

3.2 Entry Methods

In CHARM++, chares, groups and nodegroups communicate using remote method invocation. These "remote entry" methods may either take marshalled parameters, described in the next section; or special objects called messages. Messages are lower level, more efficient, more flexible, and more difficult to use than parameter marshalling.

An entry method is always a part of a chare– there are no global entry methods in CHARM++. Entry methods are declared in the the interface file as:

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

Parameters is either a list of marshalled parameters, (e.g., "int i, double x"), or a message description (e.g., "MyMessage *msg"). See section 3.3 and section 3.4 for details on these types of parameters.

Entry methods typically do not return data– in C++, they have return type "void". An entry method with the same name as its enclosing class is a constructor. Constructors in C++have no return type. Finally, sync methods, described below, may return a message.

3.2.1 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 give an entry method: threaded, sync, exclusive, nokeep, notrace, immediate, expedited.

Threaded entry methods are simply entry methods which are run in their own nonpremptible threads. To make an entry method threaded, one simply adds the keyword threaded to the attribute list of that entry method.

Sync entry methods are special in that calls to sync entry methods are blocking - they do not return control to the caller until the method is finished executing completely. Sync methods may have return values; however, they may only return messages. To make an entry method a sync entry method, add the keyword sync to the attribute list of that entry method.

Exclusive entry methods, which exist only on node groups, are entry methods that do not execute while other exclusive entry methods of its node group are executing in the same node. If one exclusive method of a node group is executing on node 0, and another one is scheduled to run on that same node, the second exclusive method will wait for the first to finish before it executes. To make an entry method exclusive, add the keyword exclusive to that entry method's attribute list.

Nokeep entry methods tells Charm++ that messages passed to these user entry methods will not be kept by the calls. Charm++ runtime may be able to adopt optimization for reusing the message memory.

Notrace entry methods simply tells Charm++ that calls to these entry methods should be not traced in trace projections or summary mode.

Immediate entry methods are entry functions in which short messages can be executed in an "immediate" fashion when they are received either by an interrupt (Network version) or by a communication thread (in SMP version). Such messages can be useful for implementing multicasts/reductions as well as data lookup, in which case processing of critical messages won't be delayed (in the scheduler queue) by entry functions that could take long time to finish. Immediate messages are only available for nodegroup entry methods. Immediate messages are implicitly "exclusive" on each node, that is one execution of immediate message will not be interrupted by another. Function CmiProbeImmediateMsg() can be called in users code to probe and process immediate messages periodically.

Expedited entry methods are entry functions that skip Charm++'s priority-based message queue. It is useful for messages that require prompt processing however in the situation when immediate message does not apply. Compared with immediate message, it provides a more general solution that works for all Charm++ objects, i.e. Chare, Group, NodeGroup and Chare Array. However, skipscheduler message still needs to be scheduled in low level Converse message queue and be processed in the order of arrival. It may still suffer from long running entry methods.

3.3 Parameter Marshalling

$\boxed{\beta}$

In CHARM++, chares, groups and nodegroups communicate by invoking each others methods. The methods may either take several parameters, described here; or take a special message object as described in the next section. Since parameters get marshalled into a message before being sent across the network, in this manual we use "message" to mean either a literal message object or a set of marshalled parameters.

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 remote chare.

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.

3.3.1 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 3.16 for more details on these routines.

```
//Declarations:
class point3d {
public:
    double x,y,z;
    void pup(PUP::er &p) {
      p|x; p|y; p|z;
    }
};

typedef struct {
    int refCount;
    char data[17];
} refChars;
PUPbytes(date);

class date {
public:
    char month,day;
    int year;
    //...non-virtual manipulation routines...
};
inline void operator|(PUP::er &p,date &d) {
    p|d.month; p|d.day;
    p|d.year;
}

//In the .ci file:
    entry void pointRefOnDate(point3d &p,refChars r[d.year],date &d);
```

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 (as shown) than by value.

For efficiency, arrays (like *refChars* above) 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 easiest to pass via marshalling, because marshalling uses the high-level pup framework.

3.4 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. In addition, messages are easier to queue and store on the receive side.

The largest difference between parameter marshalling and messages is that entry methods *keep* the messages passed to them. Thus each entry method must be passed a *new* message. On the receiving side, the entry method must either store the passed message or explicitly *delete* it, or else the message will never be destroyed, wasting memory.

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 to be valid when messages are sent across processors, with some additional coding. Also available is a mechanism for assigning *priorities* to messages that applies all kinds of messages. A detailed discussion of priorities appears later in this section.

Like all other entities involved in asynchronous method invocation, messages need to be declared in the `.ci` file. In the `.ci` file (the interface file), a message is declared as:

```
 message MessageType;
```

A message that contains variable length arrays is declared as:

```
 message MessageType {
   type1 var_name1[];
   type2 var_name2[];
   type3 var_name3[];
};
```

If the name of the message class is *MessageType*, the class must inherit publicly from a class whose name is *CMessage_MessageType*. This class is generated by the charm translator. Then message definition has the form:

```
 class MessageType : public CMessage_MessageType {
    // List of data and function members as in C++
 };
```

3.4.1 Message Creation and Deletion

Messages are allocated using the C++ `new` operator:

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

The optional arguments to the new operator are used when allocating messages with variable length arrays or prioritized messages. *sz1, sz2, ...* denote the size (in appropriate units) of the memory blocks that need to be allocated and assigned to the pointers that the message contains. The *priobits* argument denotes the size of a bitfield (number of bits) that will be used to store the message priority.

For example, to allocate a message whose class declaration is:

```
class Message : public CMessage_Message {
  // .. fixed size message
  // .. data and method members
};
```

do the following:

```
Message *msg = new Message;
```

To allocate a message whose class declaration is:

```
class VarsizeMessage : public CMessage_VarsizeMessage {
 public:
  int *firstArray;
  double *secondArray;
};
```

do the following:

```
VarsizeMessage *msg = new (10, 20) VarsizeMessage;
```

This allocates a *VarsizeMessage*, in which *firstArray* points to an array of 10 ints and *secondArray* points to an array of 20 doubles. This is explained in detail in later sections.

To add a priority bitfield to this message,

```
VarsizeMessage *msg = new (10, 20, sizeof(int)*8) VarsizeMessage;
```

Note, you must provide number of bits which is used to store the priority as the *priobits* parameter. The section on prioritized execution describes how this bitfield is used.

In Section 3.4.3 we explain how messages can contain arbitrary pointers, and how the validity of such pointers can be maintained across processors in a distributed memory machine.

When a message is sent to a chare, the programmer relinquishes control of it; the space allocated to the message is freed by the system. When a message is received at an entry point it is not freed by the runtime system. It may be reused or deleted by the programmer. Messages can be deleted using the standard C++ `delete` operator.

There are no limitations of the methods of message classes except that the message class may not redefine operators `new` or `delete`.

3.4.2 Messages with Variable Length Arrays

An ordinary message in CHARM++ is a fixed size message that is allocated internally with an envelope which encodes the size of the message. Very often, the size of the data contained in a message is not known until runtime. One can use packed messages to alleviate this problem. However, it requires multiple memory allocations (one for the message, and another for the buffer.) This can be avoided by making use of a *varsize* message. In *varsize* messages, the space required for these variable length arrays is allocated with the message such that it is contiguous to the message.

Such a message is declared as

```
 message mtype {
   type1 var_name1[];
   type2 var_name2[];
   type3 var_name3[];
 };
```

in CHARM++ interface file. The class *mtype* has to inherit from *CMessage_mtype*. In addition, it has to contain variables of corresponding names pointing to appropriate types. If any of these variables (data members) are private or protected, it should declare class *CMessage_mtype* to be a "friend" class. Thus the *mtype* class declaration should be similar to:

```
class mtype : public CMessage_mtype {
 private:
   type1 *var_name1;
   type2 *var_name2;
   type3 *var_name3;
   friend class CMessage_mtype;
};
```

An Example

Suppose a CHARM++ message contains two variable length arrays of types `int` and `double`:

```
class VarsizeMessage: public CMessage_VarsizeMessage {
  public:
    int lengthFirst;
    int lengthSecond;
    int* firstArray;
    double* secondArray;
    // other functions here
};
```

Then in the `.ci` file, this has to be declared as:

```
message VarsizeMessage {
  int firstArray[];
  double secondArray[];
};
```

We specify the types and actual names of the fields that contain variable length arrays. The dimensions of these arrays are NOT specified in the interface file, since they will be specified in the constructor of the message when message is created. In the `.h` or `.C` file, this class is declared as:

```
class VarsizeMessage : public CMessage_VarsizeMessage {
  public:
    int lengthFirst;
    int lengthSecond;
    int* firstArray;
    double* secondArray;
    // other functions here
};
```

The interface translator generates the *CMessage_VarsizeMessage* class, which contains code to properly allocate, pack and unpack the *VarsizeMessage*.

One can allocate messages of the *VarsizeMessage* class as follows:

```
// firstArray will have 4 elements
// secondArray will have 5 elements
VarsizeMessage* p = new(4, 5, 0) VarsizeMessage;
p->firstArray[2] = 13;     // the arrays have already been allocated
p->secondArray[4] = 6.7;
```

Another way of allocating a varsize message is to pass a *sizes* in an array instead of the parameter list. For example,

```
int sizes[2];
sizes[0] = 4;               // firstArray will have 4 elements
sizes[1] = 5;               // secondArray will have 5 elements
VarsizeMessage* p = new(sizes, 0) VarsizeMessage;
p->firstArray[2] = 13;     // the arrays have already been allocated
p->secondArray[4] = 6.7;
```

No special handling is needed for deleting varsize messages.

3.4.3 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.

3.4.4 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 kernel
  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.

3.4.5 Prioritized Execution

By default, Charm++ will process the messages you send in roughly FIFO order. For most programs, this behavior is fine. However, some programs need more explicit control over the order in which messages are processed. Charm++ allows you to control queueing behavior on a per-message basis.

The simplest call available to change the order in which messages are processed is CkSetQueueing.
void CkSetQueueing(MsgType message, int queueingtype)

where *queueingtype* is one of the following constants:

```
  CK_QUEUEING_FIFO
  CK_QUEUEING_LIFO
  CK_QUEUEING_IFIFO
  CK_QUEUEING_ILIFO
  CK_QUEUEING_BFIFO
  CK_QUEUEING_BLIFO
```

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);
```

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.

The other four options involve the use of priorities. 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 3.4.1. 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:
unsigned int *CkPriorityPtr(MsgType msg)

There are two kinds of priorities which can be attached to a message: *integer priorities* and *bitvector priorities*. Integer priorities are quite straightforward. One allocates a message, setting aside enough space (in bits) in the message to hold the priority, which is an integer. 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;
*CkPriorityPtr(msg) = prio;
CkSetQueueing(msg, CK_QUEUEING_IFIFO);
```

The predefined constant CK_QUEUEING_IFIFO indicates that the message contains an integer priority, and that if there are other messages of the same priority, they should be sequenced in FIFO order (relative to each other). Similarly, a CK_QUEUEING_ILIFO is available. Note that MAXINT is the lowest priority, and **NEGATIVE_MAXINT** is the highest priority.

Bitvector priorities are somewhat more complicated. 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 $.001001_{\mathrm{binary}}$. As with the simpler kind of priority, higher numbers represent lower priorities. Unlike the simpler kind of priority, bitvectors can be of arbitrary length, therefore, 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 to 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.

A final warning about prioritized execution: CHARM++ always processes messages in *roughly* the order you specify; it never guarantees to deliver the messages in *precisely* the order you specify. However, it makes a serious attempt to be "close", so priorities can strongly affect the efficiency of your program.

3.4.6 Immediate Messages

Immediate messages are specical 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. Also see Section 3.2.1 and example in *charm/pgms/charm++/megatest/immediatering.C*.

3.5 Chare Objects

Chares are concurrent objects with methods that can be invoked remotely. These methods are known as entry methods, and must be specified in the interface (`.ci`) file:

```
chare ChareType
{
    entry   ChareType             (parameters1);
    entry   void EntryMethodName2 (parameters2);
};
```

A corresponding chare definition in the `.h` file would have the form:

```
class ChareType : public Chare [: superclass names] {
     // Data and member functions as in C++
     // One or more entry methods definitions of the form:
public:
   ChareType(parameters2)
      { // C++ code block  }
   void EntryMethodName2(parameters2)
      { // C++ code block  }
};
```

Chares are concurrent objects encapsulating medium-grained units of work. 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[11]. 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[12].

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) and they must have exactly one argument which is a pointer to a message.

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

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

Each chare instance is identified by a *handle* which is essentially a global pointer, and is unique across all processors. The handle of a chare has type CkChareID. The variable thishandle holds the handle of the chare whose entry function or public function is currently executing. thishandle is a public instance variable of the chare object (it is inherited from the system-defined superclass for chares, Chare). thishandle can be used to set fields in a message. This mechanism allows chares to send their handles to other chares.

3.5.1 Chare Creation

First, a chare needs to be declared, both in `.ci` file and in `.h` file, as stated earlier. The following is an example of declaration for a chare of user-defined type *C*, where *M1* and *M2* are user-defined message types, and *someEntry* is an entry method.

In the `mod.ci` file we have:

```
module mod {
  chare C {
    entry C(parameters);
    entry void someEntry(parameters);
  };
}
```

and in the `mod.h` file:

```
#include "mod.decl.h"
class C : public Chare {
  public:
    C(parameters);
    void someEntry(parameters);
};
```

Now one can use the class CProxy_*chareType* to create a new instance of a chare. Here *chareType* gets replaced with whatever chare type we want. For the above example, proxies would be of type CProxy_*C*. A number of chare creation calls exist as static or instance methods of class CProxy_*chareType*:

```
CProxy_chareType::ckNew(parameters, CkChareID *vHdl, int destPE);
```

Each item above is optional, and:

- *chareType* is the name of the type of chare to be created.
- *parameters* must correspond to the parameters for the constructor entry method. If the constructor takes void, pass nothing here.
- *vHdl* is a pointer to a chare handle of type CkChareID, which is filled by the ckNew method. This optional argument can be used if the user desires to have a *virtual* handle to the instance of the chare that will be created. This handle is useful for sending messages to the chare, even though it has not yet been created on any processor. Messages sent to this virtual handle are either queued up to be sent to the chare after it has been created, or simply redirected if the chare has already been created. For performance reasons, therefore, virtual handles should be used only when absolutely necessary. Virtual handles are otherwise like normal handles, and may be sent to other processors in messages.
- *destPE*: when a chare is to be created at a specific processor, the *destPE* is used to specify that processor. Note that, in general, for good load balancing, the user should let CHARM++ determine the processor on which to create a chare. Under unusual circumstances, however, the user may want to choose the destination processor. If a process replicated on every processor is desired, then a chare group should be used. If no particular processor is required, the parameter can be omitted, or CK_PE_ANY.

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. When a chare is created, it is initialized by calling its constructor entry method with the message parameter specified to the chare creation method. The method operator does not return any value but fills in the virtual handle to the newly created chare if specified.

The following are some examples on how to use the chare creation method to create chares.

1. This will create a new chare of type *C* on *any* processor:

   ```
   CProxy_C chareProxy = CProxy_C::ckNew(parameters);
   ```

2. This will create a new chare of type *C* on processor destPE:

   ```
   CProxy_C chareProxy = CProxy_C::ckNew(parameters, destPE);
   ```

3. The following first creates a CkChareID *cid*, then creates a new chare of type *C* on processor *destPE*:

   ```
   CkChareID cid;
   CProxy_C::ckNew(parameters, &cid, destPE);
   CProxy_C chareProxy(cid);
   ```

3.5.2 Method Invocation on Chares

A message may be sent to a chare 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.

3.5.3 Local Access

$\boxed{\beta}$

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 if the chare does not exist or is on another processor.

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

3.6 Read-only Variables, Messages and Arrays

Since Charm++ does not allow global variables for keeping programs portable across a wide range of machines, it provides a special mechanism for sharing data amongst all objects. *Read-only* variables, 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 `main::main()` function. They can be accessed from any chare on any processor as "global" variables.When a variable is declared as read only,it is PUPped so that it can be accessed from any chare on any processor. Large data structures containing pointers can be made available as read-only variables using read-only messages or read-only arrays. Read-only variables, messages and arrays can be used just like local variables for each processor, but the user has to allocate space for read-only messages using new to create the message in the main function of the mainchare.

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

```
 readonly Type ReadonlyVarName;
```

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

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

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

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

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

Read-only variables, messages and arrays must be declared either as global or as public class static data, 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.

3.7 Basic Arrays

Arrays are arbitrarily-sized collections of chares. 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. 1D array), several integers (i.e. a multidimensional array), or an arbitrary string of bytes (e.g. a binary tree index).

Array elements can be dynamically created and destroyed on any processor, and messages for the elements will still arrive properly. Array elements can be migrated at any time, allowing arrays to be efficiently load balanced. Array elements can also receive array broadcasts and contribute to array reductions.

3.7.1 Declaring a 1D Array

You can declare a one-dimensional chare array as:

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

Just as every Chare inherits from the system class Chare, every array element inherits from the system class ArrayElement (or one of its subclasses, ArrayElement1D, ArrayElement2D, ArrayElement3D, ArrayElement4D, ArrayElement5D, or ArrayElement6D). Just as a Chare inherits "thishandle", each array element inherits "thisArrayID", the CkArrayID of its array, and "thisIndex", the element's array index.

```
class A : public ArrayElement1D {
  public:
    A(parameters1);
    A(CkMigrateMessage *);

    void someEntry(parameters2);
};
```

Note *A*'s odd migration constructor, which is normally empty:

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

Read the section "Migratable Array Elements" for more information on the CkMigrateMessage constructor.

3.7.2 Creating a Simple Array

You always create an array using the CProxy_Array::ckNew routine. This returns a proxy object, which can be kept, copied, or sent in messages. To create a 1D array containing elements indexed (0, 1, ..., *num_elements*-1), use:

```
CProxy_A1 a1 = CProxy_A1::ckNew(parameters,num_elements);
```

The constructor is invoked on each array element. For creating higher-dimensional arrays, or for more options when creating the array, see section 3.8.2.

3.7.3 Messages

An array proxy responds to the appropriate index call– for 1D arrays, use [i] or (i); for 2D use (x,y); for 3D use (x,y,z); and for user-defined types use [f] or (f).

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– by default, the system will buffer the message until the element is created[13].

Messages are not guarenteed to be delivered in order. For example, if I invoke a method A, then method B; it is possible for B to be executed before A.

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

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

3.7.4 Broadcasts

To broadcast a message to all the current elements of an array, simply omit the index, as:

```
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.

[13]However, the element must eventually be created– i.e., within a 3-minute buffering period.

3.7.5 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 on the elements of a Chare array.

The data to be reduced comes from each array element, which must call the contribute method:

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

Reductions are described in more detail in Section 3.14.

3.7.6 Destroying Arrays

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();
```

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.

3.8 Advanced Arrays

The basic array features described above (creation, messaging, broadcasts, and reductions) are needed in almost every Charm++ program. The more advanced techniques that follow are not universally needed; but are still often useful.

3.8.1 Declaring Multidimensional, or User-defined Index Arrays

Charm++ contains direct support for multidimensional and even user-defined index arrays. These arrays can be declared as:

```
//In the .ci file:
message MyMsg;
array [1D] A1 { entry A1(); entry void e(parameters);}
array [2D] A2 { entry A2(); entry void e(parameters);}
array [3D] A3 { entry A3(); entry void e(parameters);}
array [4D] A4 { entry A4(); entry void e(parameters);}
array [5D] A5 { entry A5(); entry void e(parameters);}
array [6D] A6 { entry A6(); entry void e(parameters);}
array [Foo] AF { entry AF(); entry void e(parameters);}
```

The last declaration expects an array index of type CkArrayIndex*Foo*, which must be defined before including the `.decl.h` file (see “User-defined array index type” below).

```
//In the .h file:
class A1:public ArrayElement1D { public: A1(){} ...};
class A2:public ArrayElement2D { public: A2(){} ...};
class A3:public ArrayElement3D { public: A3(){} ...};
class A4:public ArrayElement4D { public: A4(){} ...};
class A5:public ArrayElement5D { public: A5(){} ...};
class A6:public ArrayElement6D { public: A6(){} ...};
class AF:public ArrayElementT<Foo> { public: AF(){} ...};
```

A 1D array element can access its index via its inherited “thisIndex” field; a 2D via “thisIndex.x” and “thisIndex.y”, and a 3D via “thisIndex.x”, “thisIndex.y”, and “thisIndex.z”. The subfields of 4D, 5D, and 6D are respectively {w,x,y,z}, {v,w,x,y,z}, and {x1,y1,z1,x2,y2,z2}. A user-defined index array can access its index as “thisIndex”.

3.8.2 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 your 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.

3.8.3 Advanced Array Creation: 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.

3.8.4 Advanced Array Creation: Map Object

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" processor of each element. The home processor is the processor responsible for maintaining the location of the the element.

There is a default map object, which maps 1D array indices in a round-robin fashion to processors, and maps other array indices based on a hash function.

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(int numInitialElements,CkArrayID aid);
  //Return the home processor number for this element of this array
  virtual int procNum(int arrayHdl,const CkArrayIndex &element);
}
```

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);
```

3.8.5 Advanced Array Creation: 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.

3.8.6 Advanced Array Creation: Bound Arrays

$\boxed{\beta}$

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);
```

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 guarenteed to always be able to interact using ckLocal (see section 3.8.12), although the local pointer must be refreshed after any migration.

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.

3.8.7 Advanced Array Creation: 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 the reduction manager (see “Array Reductions”) and load balancer (see 3.11.1)– 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 procesor to create an array element on based on the current map object.

3.8.8 Advanced Array Creation: Demand Creation

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 paramters) 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.

3.8.9 User-defined array index type

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 ArrayElementT<Foo>
{ 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 ArrayElementT, a class templated by the index type *Foo*. 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);
}
```

3.8.10 Migratable Array Elements

Array objects can migrate from one PE to another. For example, the load balancer (see section 3.11.1) might migrate array elements to better balance the load between processors. For an array element to migrate, it must implement a pack/unpack or “pup” method:

```
//In the .h file:
class A2:public ArrayElement2D {
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)
{
    ArrayElement2D::pup(p); //<- MUST call superclass's pup routine
    p|nt;
    p|chr;
    p(flt,7);
    p|numDbl;
    if (p.isUnpacking()) dbl=new double[numDbl];
    p(dbl,numDbl);
}
```

Please note that if your object contains Structured Dagger code (see section “Structured Dagger”) you must use the following syntax to correctly pup the object:

```
class bar: public ArrayElement3D {
 private:
    int a,b;
 public:
    bar_SDAG_CODE
    ...other methods...

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

See the section "PUP" for more details on pup routines and the PUP::er type.

The system uses one pup routine to do both packing and unpacking by passing different types of PUP::ers to it. You can determine what type of PUP::er has been passed to you with the isPacking(), isUnpacking(), and isSizing() calls.

An array element can migrate by calling the migrateMe(*destination processor*) member function– this call must be the last action in an element entry point. The system can also migrate array elements for load balancing (see the section 3.11.3).

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 (killing off 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.

3.8.11 Load Balancing Chare Arrays

see section 3.11.1

3.8.12 Local Access

$\boxed{\beta}$

You can 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.

3.8.13 Array Section

$\boxed{\beta}$

Charm++ now supports array section. Array section is a subset of array elements in a chare array. You can create a special proxy for an array section and do multicast using the proxy. 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, it can be plugged in as a delegation module.

For each chare array "A" declared in a ci file, the definition of section proxy of type "CProxySection_A" is automatically generated in the decl and def header files. You can create an array section proxy in your application by invoking ckNew() to CProxySection:

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

Once you have the array section proxy, you can do multicast to all the section members, or send messages to one member using its index that is local to the section, like these:

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

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

In the multicast example above, for a section with k members, total number of k messages will be sent to all the memebers, which is considered 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. This library also supports section based reduction.

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
cd charm/net-linux/tmp
make multicast

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

CkMulticast library is implemented using delegation(Sec. 3.20). A special "CkMulticastMgr" Chare Group is created as a deletegation 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, you need to create the CkMulticastMgr Group first, and setup the delegation relationship between the section proxy and CkMulticastMgr Group. You only need to create one CkMulticastMgr Group though, it can serve as multicast/reduction delegation for all array sections:

```
CProxySection_Hello sectProxy = CProxySection_Hello::ckNew(...);
CkGroupID mCastGrpId = CProxy_CkMulticastMgr::ckNew();
CkMulticastMgr *mcastGrp = CProxy_CkMulticastMgr(mCastGrpId).ckLocalBranch();

sectProxy.ckSectionDelegate(mCastGrpId);  // initialize section proxy

sectProxy.someEntry(...)           //multicast via delegation library as before
```

Note, to use CkMulticast library, all multicast messages must inherit from CkMcastBaseMsg, as following:

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

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

Array Section Reduction Since an array element can be members for multiple array sections, there has to be a way for each array element to tell for which array section it wants to contribute. For this purpose, a data structure called "CkSectionInfo" is created by CkMulticastMgr for each array section that the array element belongs to. When doing section reduction, the array element needs to pass the CkSectionInfo as a parameter in the contribute(). The CkSectionInfo can be retrieved from a message in a multicast entry function 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 cookie includes some context sensive information for example the reduction counter. Function CkGetSectionInfo() only update some part of the data in cookie, not creating a brand new one.

Similar to array reduction, to use section based reduction, a reduction client CkCallback object need to be created. 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. See the followin 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 by 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(...));
```

Same as in array reduction, users can use built-in reduction types(Section 3.14.2) or define his/her own reducer functions (Section 3.14.3).

Array section multicast/reduction when migration happens Using multicast/reduction, you don't need to worry about array migrations. When migration happens, you can still use the CkSectionInfo for reduction. Reduction messages will be correctly delivered but may not be as efficient until a new multicast spanning tree is updated internally in CkMulticastMgr. A new updated CkSectionInfo is always contained in a multicast message, so it is recommended that CkGetSectionInfo() function is called everytime time when a multicast message arrives(as shown in the above SayHi example).

3.9 Group Objects

A group[14] is a collection of chares where there exists one chare (or *branch*) on each processor. Each branch has its own data members. Groups have a definition syntax similar to normal chares, except that they must inherit from the system defined class Group, rather than Chare.

In the interface file, we declare

```
group GroupType {
  // Interface specifications as for normal chares
};
```

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

```
class GroupType : public Group [,other superclasses ] {
 // Data and member functions as in C++
 // Entry functions as for normal chares
};
```

A group is identified by a globally unique group identifier, whose type is CkGroupID. This identifier is common to all of the group's branches and can be obtained from the variable thisgroup, which is a public local variable of the Group superclass. For groups, thishandle is the handle of the particular branch in which the function is executing: it is a normal chare handle.

Groups can be used to implement data-parallel operations easily. In addition to sending messages to a particular branch of a group, one can broadcast messages to all branches of a group. There can be many instances corresponding to a group type. Each instance has a different group handle, and its own set of branches.

3.9.1 Group Creation

Given a `.ci` file as follows:

```
group G {
  entry G(parameters1);
  entry void someEntry(parameters2);
};
```

and the following `.h` file:

```
class G : public Group {
  public:
    G(parameters1);
    void someEntry(parameters2);
};
```

we can create a group in a manner similar to a regular chare.

```
CProxy_G groupProxy = CProxy_G::ckNew(parameters1);
or
CkGroupID groupId = CProxy_G::ckNew(parameters1);
CProxy_G groupProxy(groupId);
```

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

3.9.2 Method Invocation on Groups

Before sending a message to a group via an entry method, we need to get a proxy of that group.

A message may be sent to a particular branch of group using the notation:

```
groupProxy[Processor].EntryMethod(parameters);
```

This sends the given parameters to the branch of the group referred to by *groupProxy* which is on processor number *Processor* at the entry method *EntryMethod*, which must be a valid entry method of that group type. 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 processors) using the notation :

```
groupProxy.EntryMethod(parameters);
```

This sends the given parameters to all branches of the group at the entry method *EntryMethod*, which must be a valid entry method of that group type. This call is asynchronous and non-blocking; it returns immediately after sending the message.

Sequential objects, chares and other groups can gain access to the local (i.e., on their processor) group object using:

```
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.

Thus a dynamically created chare can call a public method of a group without needing to know which processor it actually resides: the method executes in the local branch of the group.

One very nice use of Groups is to reduce the number of messages sent between processors by collecting the data from all the chares on a single processor before sending that data to the mainchare. To do this, create basic chares to break up the work of a problem. Also, 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 one processor are complete, the local branch of the group can then report to the main chare. This reduces the number of messages sent to main from the number of chares created to the number of processors.

3.10 Nodegroup Objects

Node groups are very similar to the group objects already discussed in that node groups are collections of chares as well. Node groups, however, have one chare per node rather than one chare per processor. So, each node contains a branch of the node group, each containing one set of data members. When an entry method of a node group is executed, it runs on only one processor within each node.

Node groups have a definition syntax that is very similar to groups. Rather than inheriting from the system defined class, Group, node groups inherit from Nodegroup. 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 Nodegroup [,other superclasses ] {
 // 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 like with groups, this identifier is common to all branches of the nodegroup and can be obtained from the variable thisgroup, and once again, thishandle is the handle of the particular branch in which the function is executing.

Node groups may possess exclusive entry methods. These are entry methods that will not run while other other exclusive entry methods of that node group are running on the same node. For instructions for making an entry method exclusive, refer to section 3.2.1.

For certain applications, node groups can be used in the place of regular groups to cut down on messaging overhead when shared memory access is possible. For example, consider a parallel program that does one calculation that can be decomposed into several mutually exclusive subcalculations. The program distributes the work amongst all of the processors, the subresults are all stored in the local branch of a group, and when the local branch has recieved all of its results, it relays everything to one particular processor where the subresults are put together into the final result. When normal groups are used, the number of messages sent is $O(\#\text{ of processors})$. However, if node groups are used, a number of message sends will be replaced by local memory accesses if there is more than one processor per node. Instead, the number of messages sent is $O(\#\text{ of nodes})$.

Just like groups, there can be many instances corresponding to a single node group type, and each instance has a different group handle, and its own set of branches.

3.10.1 Method Invocation on Nodegroups

Methods can be invoked either on a particular branch of a nodegroup by specifying a *node number* as a method parameter. In the absence of such a parameter, the call is treated as broadcast on a nodegroup, i.e. executed by all nodes. When a method is invoked on a particular branch of a nodegroup, it may be executed by *ANY* processor in that node. Thus two invocations of a specific method on a particular branch of a nodegroup may be carried out *simultaneously* by two different processors of the node. If that method contains code that should be executed by only one processor at a time, the method should be flagged exclusive in the interface file. If a method M of a nodegroup NG is marked exclusive, it means that while that method is being executed by any processor within a node, no other processor within the same node may execute any other *exclusive* method of that nodegroup branch. Other processors are free to execute other *non-exclusive* methods of that nodegroup branch, however.

The local branch of a nodegroup can be accessed using CkLocalNodeBranch() function. Thus data members could be accessed/modified or methods could be invoked on a branch of a nodegroup using this function. Note that such accesses are *not* thread-safe by default. Concurrent invocation of a method on a nodegroup by different processors within a node may result in unpredictable runtime behavior. One way to avoid this is to use node-level locks (described in Converse manual.)

CkLocalNodeBranch returns a generic (`void *`) pointer, similar to CkLocalBranch. Also, the static method ckLocalNodeBranch of the proxy class of appropriate nodegroup can be called to get the correct type of pointer.

3.11 Load Balancing

Charm++ supports Load Balancing, enabled by the fact there are a large number of chares or chare-array-elements typically available to map to existing processors, and that they can be migrated at runtime.

Many parallel applications, especially physical simulations, are iterative in nature. They 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 time, even in dynamic applications. In such cases, recent past is a good predictor of 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 future.

For applications without such iterative structure, or with iterative structure but without the 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 '80s. 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 (noise due to OS interrupts on small time steps, or time-shared desk-top machines), one can use a combination of the two kinds of strategies. The base-line 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.

3.11.1 Measurement-based Object Migration Strategies

In Charm++, objects(except groups, nodegroups) can migrate from processor to processor at run-time. 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 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 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.

Here are the two terms often used in Charm++ load balancing framework:

- 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. Two kinds of schemes are implemented, they are (a) centralized load balancers and (b) distributed load balancers.

3.11.2 Available Load Balancing Strategies

Load balancing can be performed in either a centralized or distributed 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 machine, it tends to be more accurate.

In distributed approaches, machine states are 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.

Listed below are the available centralized load balancers and their brief descriptions:

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

- **RefineCommLB**: Same idea as in RefineLB, but take communication into account;

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

- **RecBisectBfLB**: Recursively partition with Breadth first enumeration;

- **MetisLB**: Use Metis(tm) to partitioning object communication graph;

- **GreedyLB**: Use greedy algorithm, always pick the heaviest object to the least loaded processor.

- **GreedyCommLB**: Greedy algorithm which also takes communication graph into account;
- **ComboCentLB**: A special load balancer that can be used to combine any number of above centralized load balancers;

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 interferes with other's use of the desktop.

User can choose any load balancing strategy he or she thinks is the good for the application. The compiler and run-time options are described in section 3.11.6.

3.11.3 Load Balancing Chare Arrays

Load balancing framework is well integrated with Chare array implementation - when a chare array is created, it automatically registers its elements to load balancing framework, instrument 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 3.11.2 for a description of available load balancing strategies).

We implemented three methods 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**, **automatic with Sync mode** and **manual mode**.

In *periodic load balancing mode*, a user just needs to specify how often he wants the load balancing to occur, using +LBPeriod runtime option to specify a time interval.

In *sync mode*, users can tell load balancer explicitly when is a good time to trigger load balancing by inserting a function call in the user code.

In the above two load balancing modes, users don't need to worry about how to start load balancing. However, in one scenario, the above automatic load balancers will fail to work - array elements are created by dynamic insertion. This is because the above two load balancing modes require an application to have fixed 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, users have to use the *manual mode* to trigger load balancer themselves. The API is described below.

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

1. **periodical load balancing mode**: By default, the 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. In the default setting, load balancing happens whenever the array elements are ready with interval of 1 second. It is desirable for the application to set a larger interval using +LBPeriod runtime option. For example "+LBPeriod 5" to start load balancing roughly every 5 seconds.
2. **automatic with Sync**: Using the AtSync method, elements can only be migrated at certain points in the execution when user calls AtSync(). For the AtSync method, set usesAtSync to CmiTrue in your array element constructor. When an element is ready to migrate, call AtSync() [15]. 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 element.

   Note that *AtSync()* is not a blocking call, it just give 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

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

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. The most commonly used approach is to force the object to be idle until load balancing finishes, the object can start working again when ResumeFromSync() is called.

3. **manual mode**: The load balancer can be programmed to be started manually. To switch to the manual mode, you should call *TurnManualLBOn()* on every processor to prevent 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 3.18. 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 at the background.

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

3.11.4 Migrating objects

Load balancers migrate objects automatically. For an array element to migrate, user can refer to section 3.8.10 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 scheme where user explicitly control when the load balancing happens, user may choose to pack only a part of the data and may skip those temporary data.

3.11.5 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 load balancer whether this object is migratable or not[16].

- **lbdb− >SetLBPeriod(double s)**: this function can be called anywhere[17] to regulate the load balancing period. 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);

// if used outside of an array element
// must be called after load balancer is initialized.
```

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

[17]except that it must be called after LBDatabase has been initialized in Charm++ init phase, so for example global variable constructor is not a good place to call it.

```
LBDatabase *lbdb = LBDatabase::Object();
lbdb->SetLBPeriod(5);
```

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

3.11.6 Compiler and run-time 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 choose one to use at run-time via command line options.

Please note that linking a load balancing module is different from activating it:

- link a LB module: is to link a Load Balancer module(library) at compile time; You can link against multiple LB libraries as candidates.
- activate a LB: is to actually ask tun-time to create a 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 run-time options:

1. **compile time options:**

   - *-module NeighborLB -module GreedyCommLB ...*
     links module NeighborLB, GreedyCommLB etc into an application, but these load balancers will remain inactive at execution time unless overriden by other runtime options.
   - *-module EveryLB*
     links special module EveryLB which includes all the charmpp built-in load balancers.
   - *-balancer GreedyCommLB*
     links load balancer GreedyCommLB and invoke this load balancer at runtime. One can specify multiple load balancers for example:
     -balancer GreedyCommLB -balancer RefineLB
     This will start first load balancing step with GreedyCommLB, and second load balancing step with RefineLB, and same RefineLB in the subsequent load balancing steps.
   - *-balancer ComboCentLB:GreedyLB,RefineLB*
     One can choose to create a new combo load balancer made of multiple load balancers. The above example specifies *one* load balancer which first applies GreedyLB algorithm followed by RefineLB algorithm.

   The list of existing load balancers are in section 3.11.2. Note: you can have multiple -module *LB options. LB modules are linked into a program, but they are not activated automatically at run-time. Using -balancer at compile time in order to activate load balancers automatically at run time. Having -balancer A implies -module A, so you don't have to write -module A again, although it does not hurt. Using EveryLB is a convenient way to link against all existing load balancers. One of the load balancer called MetisLB requires METIS library which is located at: charm/src/libs/ck-libs/parmetis/METISLib/. You need to compile METIS library by "make METIS" under charm/tmp after you compile Charm++.

2. **run-time options:**

   Run-time options are doing the same thing as compile time options, but they can override the compile time options.

   - *+balancer help*
     displays all available balancers already linked in.

- *+balancer GreedyCommLB*
  activates GreedyCommLB. Multiple load balancers can be activated at the runtime to work in sequence. (same as compile time options described above)
- *+balancer ComboCentLB:GreedyLB,RefineLB*
  same as the example in +balancer compile time option.

Note: +balancer option works only if you have already linked the load balancers module at compile time. Given +balancer with a wrong LB name will results in runtime error. When you have used -balancer A as compile time option, you don't 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. **When there is no load balancer activated**

   When you don't activate any of load balancer at compile time or run time, and your program counts on a load balancer because you use *AtSync()* and expect *ResumeFromSync()* to be called to continue, be assured that your program still can run. What happens is that a special load balancer called *NullLB* is automatically created in this case which basically does nothing but calling *ResumeFromSync()* after *AtSync()*. This is to make sure program does not hang because of *AtSync()*. The *NullLB* is smart enough to keep silent if there is any other load balancer is created.

4. **Other useful run-time options**

   There are a few other run-time options for load balancing that may be useful:

   - *+LBDebug {verbose level}*
     {verbose level} can be any positive integer number. 0 to turn off. This option asks load balancer to output more information to stdout about load balancing. The bigger the verbose level, the more verbose the output is.
   - *+LBPeriod {seconds}*
     {seconds} can be any float number. This sets the minimum period time in seconds between two consecutive load balancing steps. The default value is 1 second. That is to say a second load balancing step won't happen until after 1 second since 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 run-time.
   - *+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 use this option to force a global barrier on all processors to make sure processors can only resume computation after migrations finish on all processors.
   - *+LBOff*
     Turns off load balancing instrumentation at startup time. This call turns off the instrument of both CPU and communication usage.
   - *+LBCommOff*
     Turns off load balancing instrumentation of communication at startup time. The instrument of CPU usage is left on.

3.11.7 Load Balancing Simulation

The simulation feature of 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 to later test different load balancing strategies to see which one is suitable for the programs behaviour. 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 instrument/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 first step in the program is number 0. Negative numbers will be converted to 0.

2. *+LBDumpSteps StepsNo*
   This option specify for how many steps to dump the data to the disk. If omitted, default value is 1. The program will exit after StepsNo files are dumped.

3. *+LBDumpFile FileName*
   This option specified the base name of the file into which to dump the load balancing data. If this option is not specified, the framework uses the default file `lbdata.dat`. Since multistep is allowed, to this base name it is added the step number in the form `Filename.#`; this apply to both dump and simulation.

4. *+LBSim StepStart*
   This option instructs the framework to do the simulation during the first load balancing step. When this option is specified, the load balancing data from the file specified in the *+LBDumpFile* option, with the addition of the step number, will be read and this data will be used for the load balancing. 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 options has the same meaning of *+LBDumpSteps*, except that apply for the simulation mode. Default value is 1.

6. *+LBSimProcs*
   This option may change the number of processors target of the load balancer strategy. It may be used to test the load balancer in conditions where some processor crashes or someone 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 differ 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.

As an example, we can collect the data for a 1000 processor run of a program using:

```
./charmrun pgm +p 1000 +balancer RandCentLB +LBDump 2 +LBDumpSteps 4 +LBDumpFile dump.dat
```

This will collect data on files data.dat.2,3,4,5. Then, we can use this data to observe various centralized strategies using:

```
./charmrun pgm +balancer <Strategy to test> +LBSim 2 +LBSimSteps 4
               +LBDumpFile dump.dat [+LBSimProcs 900]
```

3.11.8 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 correct information to do a correct 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. As said, the user might provide a specific function which inherit from `LBPredictorFunction` and implement its 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\n");};
  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 reimplemented.
- `predict` is the function the model implements. For example, if the function is $y = ax + b$, the method in the implemented class should be like:

  ```
  double predict(double x, double *param) {return (param[0]*x + param[1]);}
  ```

- `print` is a debugging function and it can be reimplemented 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 function, 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 behind the computation is not known, the user can leave the system to use a predefined default function.

As seen, the function can have several parameters which will be learned during the execution of the program. For this, two parameters can be setup at command line to specify the learning behaviour:

1. *+LBPredictorWindow size*
   This parameter will specify how many statistics the load balancer will keep for keeping updated the function parameters. The greater this number is, the best approximation of the workload will be, but more memory will be 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 start using 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 callable from user program to modify the predictor. These methods are:

- `void PredictorOn(LBPredictorFunction *model);`
- `void PredictorOn(LBPredictorFunction *model,int wind);`
- `void PredictorOff();`
- `void ChangePredictor(LBPredictorFunction *model);`

3.11.9 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 load balancing scheme is used for load balancing chares only at creation time. When the chare is created on a processor, there is no movement of the chare due to the seed load balancer. The measurement based load balancer described in previous subsection perform the task of moving chares during work to achieve load balance.

Several variations of strategies have been designed and analyzed.

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 to neighbors only. The overloaded processors initiate the load balancing, where a processor 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.

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

Seed load balancers for Chares:
Seed load balancers can be directly used for load balancing Chares. The default seed load balancer that is always linked without overridden is the random seed load balancer. User can picks one of the other strategies listed above and link as a plugin module into binary as described below to override the default strategy.

Seed load balancers for Array Elements:
Seed load balancers can also be used for array elements in the same way as they are used for individual chares. Chare array is a collection of individual Chares in Charm++. Since Chare Array has its internal strategy of static mapping of individual array elements to processors using *CkArrayMap* 3.8.4 [18], a special CkArrayMap called *CldMap* must be created and passed into array creation calls to interface with seed load balancer.

For creating an empty array and then inserting chares into it, the API is as follows:

```
CkArrayOptions opt;
CkGroupID cldmapID = CProxy_CldMap::ckNew();
opt.setMap(cldmapID);
CProxy_WorkUnit arr = CProxy_WorkUnit::ckNew(param, opt);
for (int i=0; i<numChares; i++)
  arr[i].insert(param);
```

[18]by default it always distributed array elements to processors in Round-Robin fashion unless a different CkArrayMap is used

For initially populating the array with chares at time of creation the API is as follows:

```
CkArrayOptions opt(numChares);
CkGroupID cldmapID = CProxy_CldMap::ckNew();
opt.setMap(cldmapID);
CProxy_WorkUnit arr = CProxy_WorkUnit::ckNew(param, opt);
```

The details about array creation are explained in section 3.8 of the manual.

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.

3.11.10 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 Chare
{
private:
  int count;
public:
  Main(CkArgMsg* m)
  {
    /*....Initialization....*/
    mainProxy = thishandle;
    CProxy_LBExample arr = CProxy_LBExample::ckNew(nElements);
    arr.doWork();
  };

  void done(void)
  {
    count++;
    if(count==nElements){
      CkPrintf("All done\n");
      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"
```

3.12 Advanced Load Balancing

3.12.1 Write a Measurement-based Object Migration Strategy

Charm++ programmers can pick load balancing strategy from Charm++'s built-in strategies(see 3.11.2) for the best performance based on the characteristics of their applications, they can also choose to write their own load balancing strategies.

Charm++ load balancing framework provides a simple scheme to incorporate new load balancing strategies. To write a new load balancing strategy involves the following steps (We use an example of writing a centralized load balancer *fooLB* to illustrate the steps).

1. Create files named *fooLB.ci, fooLB.h and fooLB.C.* One can choose to copy and rename the files RandCentLB (a random centralized load balancer) and rename the class name in those files.
2. Implement the *fooLB* class method — **fooLB::work(CentralLB::LDStats* stats, int count)** This method takes the load balancing database (*stats*) as an input, and output the new mapping of objects to processors in *stats-¿to_proc* array.
3. To compile the strategy files, first add *fooLB* into the load balancer list in charm/tmp/Makefile_lb.sh. Run the script in charm/tmp, which creates the new Makefile namded "Make.lb".
4. 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.

3.12.2 Understand Load Balancing Database Data Structure

To write a load balancing strategy, one may want 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
    double total_walltime;
    double total_cputime;
    double idletime;
    double bg_walltime;
    double 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. *procs* array defines processor attributes and usage data for each processor;
2. *objData* array records per object information, *LDObjData* is defined in *lbdb.h*;
3. *commData* array records per communication information. *LDCommData* is defined in *lbdb.h*.

3.13 Quiescence Detection

In Charm++, quiescence is defined as the state in which no processor is executing an entry point, and no messages are awaiting processing.

Charm++ provides two facilities for detecting quiescence: CkStartQD and CkWaitQD.

CkStartQD registers with the system a callback that should be made the next time quiescence is detected. CkStartQD has two variants which expect the following arguments:

1. 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 *CkIndex* object corresponding to the chare containing that entry method. The syntax of this call is as follows:

   myIdx=CkIndex_*ChareName*::*EntryMethod*(*parameters*);

   where *ChareName* is the name 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.

2. A *CkCallback* object. The syntax of this call looks like:

   ```
     CkStartQD(const CkCallback& cb);
   ```

   Upon quiescence detection, specified callback is called with no parameters.

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();
```

Keep in mind that CkWaitQD should only be called from threaded entry methods because a call to CkWaitQD suspends the current thread of execution, and if it were called outside of a threaded entry method it would suspend the main thread of execution of the processor from which CkWaitQD was called and the entire program would come to a grinding halt on that processor.

3.14 Reductions

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 reduction types, below). Unlike sending a message, you may use data after the call to contribute. All members must call contribute, and all must use the same reduction type.

When you create a new member, it is expected to contribute to the next reduction not already in progress on that processor. The reduction will complete properly even if members are migrated or deleted during the reduction.

For example, if we want to sum each 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 an array has a pair of doubles *forces*[2] which need to be summed up (separately) across every element, from each element call:

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

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

The result of the reduction operation is passed to the *reduction client*. Many different kinds of reduction clients can be used, as explained below (Section 3.14.1).

3.14.1 Reduction Clients

After the data is reduced, it is passed to a you via a callback object, as described in section 3.15. The message passed to the callback is of type CkReductionMsg. 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);
```

If no member passes a callback to contribute, the reduction will use the default callback. You set the default callback for an array or group using the ckSetReductionClient proxy call on processor zero. 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 pgms/charm++/RedExample for a complete example).

For backward compatability, rather than a general callback you can specify a peculiar 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:
  myProxy.setReductionClient(myClient,(void *)NULL);

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;
}
```

3.14.2 Built-in Reduction Types

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

The first four reductions (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::sum_int, sum_float, sum_double– the result will be the sum of the given numbers.
2. CkReduction::product_int, product_float, product_double– the result will be the product of the given numbers.
3. CkReduction::max_int, max_float, max_double– the result will be the largest of the given numbers.
4. CkReduction::min_int, min_float, min_double– the result will be the smallest of the given numbers.
5. CkReduction::logical_and– the result will be the logical AND of the given integers. 0 is false, nonzero is true.
6. CkReduction::logical_or– the result will be the logical OR of the given integers.
7. 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.
8. CkReduction::concat– the result will be a byte-by-byte concatentation of all the contributed data. There is no separation added between different contributions.

CkReduction::set returns a collection of CkReduction::setElement objects, one per contribution. This class has 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. Add a source field to your contribution if you need to know which array element gave a particular contribution. This will require you to do your own serialize/unserialize operation on your element structure if your reduction element data is complex. Consider using the PUP interface see 3.16 to simplify your object serialization needs.

If your data is order dependant, 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 3.14.3) for details.

3.14.3 Defining a New Reduction Type

It is possible to define a new type of reduction, performing a user-defined operation on user-defined data. A reduction function combines separate contributions (from this or other processors) 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 integers 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);
}
```

You must register your reduction function with Charm++ using CkReduction::addReducer from an initcall routine (see section 3.18 for details on the initcall mechanism). CkReduction::addReducer returns a CkReduction::reducerType which you can later pass to contribute. Since initcall 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:

```
//In the .ci file:
  initcall void registerSumTwoShorts(void);

//In some .C file:
/*global*/ CkReduction::reducerType sumTwoShortsType;
/*initcall*/ void registerSumTwoShorts(void)
{
  sumTwoShortsType=CkReduction::addReducer(sumTwoShorts);
}

//In some member:
  short data[2]=...;
  contribute(2*sizeof(short),data,sumTwoShortsType);
```

Note that you cannot call CkReduction::addReducer from anywhere but in an initcall routine.

3.15 Callbacks

A callback is a generic way to transfer control back to a client after a library has finished. For example, after finishing a reduction, you might want the results passed to some chare's entry method. To do this, you create an object of type CkCallback with the chare's CkChareID and entry method index, then pass the callback object to the reduction library.

3.15.1 Client Interface

You can create a CkCallback object in a number of ways, depending on what you want to have happen when the callback is finally invoked. The callback will be invoked with a Charm++ message; but the message type will depend on the library that actually invokes the callback. Check the library documentation to see what kind of message the library will send to your callback. In any case, you are required to free the message passed to you via the callback.

The callbacks that go to chares require an "entry method index", an integer that identifies which entry method will be called. You can get an entry method index using the syntax:

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. An entry method index is the Charm++ version of a function pointer.

There are a number of ways to build callbacks, depending on what you want to have happen when the callback is invoked:

1. CkCallback(CkCallbackFn fn,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. 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 the 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 of the given Chare. 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. As usual, an array proxy will work just as well as 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, a CkCallback(CkCallback::resumeThread), can only be used from within threaded entry methods. This type of callback is typically hidden within a thread-capable library, so is discussed further in the library section.

3.15.2 Library Interface

Here, a "library" is simply any code which can be called from several different places. From the point of view of a library, a CkCallback is a destination for the library's result. CkCallback objects can be freely copied, marshalled, or even sent in messages.

Postponing threads for a moment, the only thing you can do with a CkCallback is to move it around or send a message to it:

```
//Main library entry point, called by asynchronous users:
void myLibrary(...library parameters...,const CkCallback &cb)
{
  ..start some parallel computation, dragging cb along...
}

//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, so you *do* need to document the type of message you will send to the callback so the user knows what to expect.

Threaded clients are a bit more complicated– you need to suspend the calling thread using "thread_delay" which, after the corresponding "send", returns the sent message to its caller. For example:

```
//Main library entry point, called by threaded users:
myLibMsg *myThreadedLibrary(...library parameters...)
{
  CkCallback cb(CkCallback::resumeThread);
  myLibrary(...,cb); //Just call normal library with new cb
  return cb.thread_delay(); //Will suspend until cb.send() is called
}
```

"thread_delay" just immediately returns NULL for non-threaded callbacks, so you can even combine the threaded and non-threaded interfaces using C++'s default parameters. For example:

```
//Main library entry point, called by threaded users:
myLibMsg *myGenericLibrary(...library parameters...,
  CkCallback cb=CkCallback(CkCallback::resumeThread))
{
  myLibrary(...,cb);
  //For threaded clients, suspends until cb.send, then returns message;
  // for non-threaded clients, just returns NULL:
  return cb.thread_delay();
}
```

3.16 PUP

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

3.16.1 PUP contract

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

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

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

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

- A simple type, including char, short, int, long, float, or double. In this case, `p|a;` copies the data in-place. This is equivalent to passing the type directly to the PUP::er using `p(a)`.
- Any object with a pup routine. In this case, `p|a;` calls the object's pup routine. This is equivalent to the statement `a.pup(p);`.
- A pointer to a PUP::able object, as described in Section 3.16.4. 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 |`.

For container types, you must simply pup each element of the container. For arrays, you can use the utility routine PUParray, which takes the PUP::er, the array base pointer, and the array length. This utility routine 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];
}
```

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".

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 3.16.3 for more information and examples.

If the object does not have a pup routine, 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;
}
```

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 routine. 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 routine. This is a direct call to pup a set of bytes.

Note that pupping as bytes is just like using 'memcpy': it does nothing to the data but 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 will prevent your pup routine from ever being able to work across different machine architectures. We don't do this very often yet, but eventually may, so pupping as bytes is currently discouraged.

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

For arrays 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.

PUP structured dagger Please note that if your object contains Structured Dagger code (see section "Structured Dagger") you must call the generated routine __sdag_pup 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...
    }
};
```

PUP modes Charm++ uses your pup routine to both pack and unpack, by passing different types of PUP::ers to it. The routine p.isUnpacking() returns true if your object is being unpacked—that is, your object's values are being restored. Your pup routine 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 routines 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() routines. 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 routine. This is normally only needed for pup routines called via the C or f90 interface, as provided by AMPI or the FEM framework. 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.

3.16.2 Life Cycle

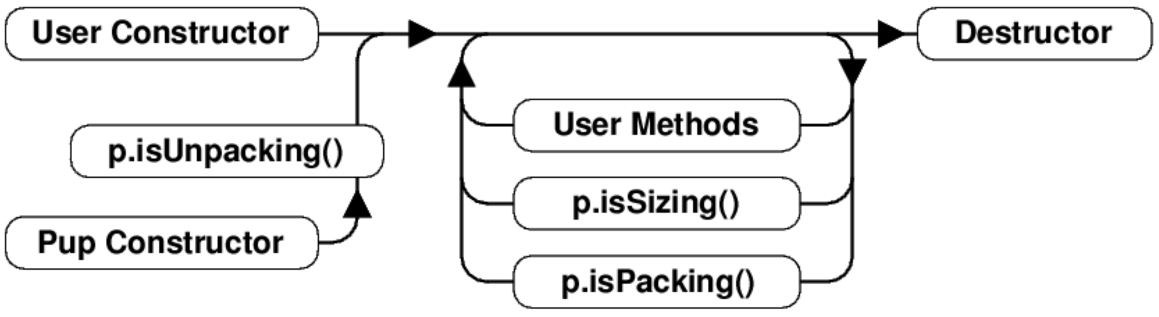

Figure 1: Life cycle of an object with a pup routine.

The life cycle of an object with a pup routine is shown in Figure 1. As usual in C++, objects are constructed, do some processing, and are then destroyed.

Objects can be created in one of two ways: they can be created using a normal constructor as usual; or they can be created using their pup constructor. The pup constructor for CHARM++ array elements and PUP::able objects is a "migration constructor" that takes a single "CkMigrateMessage *"; 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 routine 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 routine can be called in `isSizing` or `isPacking` mode. User methods and sizing or packing pup routines can be called repeatedly over the object lifetime.

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

3.16.3 Dynamic Allocation

If your class has fields that are dynamically allocated, when unpacking these need to be allocated (in the usual way) before you pup them. Deallocation should be left to the class destructor as usual.

No allocation The simplest case is when there is no dynamic allocation.

```
class keepsFoo : public mySuperclass {
private:
    foo f; /* simple foo object*/
public:
    keepsFoo(void) { }
    void pup(PUP::er &p) {
      mySuperclass::pup(p);
      p|f; // pup f's fields (calls f.pup(p);)
    }
    ~keepsFoo() { }
};
```

Allocation outside pup The next simplest case is when we contain a class that is always allocated during our constructor, and deallocated during our destructor. Then no allocation is needed within the pup routine.

```
class keepsHeapFoo : public mySuperclass {
private:
    foo *f; /*Heap-allocated foo object*/
public:
    keepsHeapFoo(void) {
      f=new foo;
    }
    void pup(PUP::er &p) {
      mySuperclass::pup(p);
      p|*f; // pup f's fields (calls f->pup(p))
    }
    ~keepsHeapFoo() {delete f;}
};
```

Allocation during pup If we need values obtained during the pup routine before we can allocate the class, we must allocate the class inside the pup routine. Be sure to protect the allocation with "if (p.isUnpacking())".

```
class keepsOneFoo : public mySuperclass {
private:
    foo *f; /*Heap-allocated foo object*/
public:
    keepsOneFoo(...) {f=new foo(...);}
    keepsOneFoo() {f=NULL;} /* pup constructor */
    void pup(PUP::er &p) {
      mySuperclass::pup(p);
      ...
      if (p.isUnpacking()) /* must allocate foo now */
         f=new foo(...);
      p|*f;//pup f's fields
    }
    ~keepsOneFoo() {delete f;}
};
```

Allocatable array For example, if we keep an array of doubles, we need to know how many doubles there are before we can allocate the array. Hence we must first pup the array length, do our allocation, and then pup the array data. We could allocate memory using malloc/free or other allocators in exactly the same way.

```
class keepsDoubles : public mySuperclass {
private:
    int n;
    double *arr;/*new'd array of n doubles*/
public:
    keepsDoubles(int n_) {
      n=n_;
      arr=new double[n];
    }
    keepsDoubles() { }

    void pup(PUP::er &p) {
      mySuperclass::pup(p);
      p|n;//pup the array length n
      if (p.isUnpacking())  arr=new double[n];
      PUParray(p,arr,n); //pup data in the array
    }

    ~keepsDoubles() {delete[] arr;}
};
```

NULL object pointer If our allocated object may be NULL, our allocation becomes much more complicated. We must first check and pup a flag to indicate whether the object exists, then depending on the flag, pup the object.

```
class keepsNullFoo : public mySuperclass {
private:
    foo *f; /*Heap-allocated foo object, or NULL*/
public:
    keepsNullFoo(...) { if (...) f=new foo(...);}
    keepsNullFoo() {f=NULL;}
    void pup(PUP::er &p) {
      mySuperclass::pup(p);
      int has_f=(f!=NULL);
      p|has_f;
      if (has_f)
        if (p.isUnpacking()) f=new foo;
        p|*f;
       else
        f=NULL;

    }
    ~keepsNullFoo() {if (f!=NULL) delete f;}
};
```

This sort of code is normally much longer and more error-prone if split into the various packing/unpacking cases.

Array of classes An array of actual classes can be treated exactly the same way as an array of basic types. PUParray will pup each element of the array properly, calling the appropriate `operator|`.

```
class keepsFoos : public mySuperclass {
private:
    int n;
    foo *arr;/*new'd array of n foos*/
public:
    keepsFoos(int n_) {
      n=n_;
      arr=new foo[n];
    }
    keepsFoos() { arr=NULL; }

    void pup(PUP::er &p) {
      mySuperclass::pup(p);
      p|n;//pup the array length n
      if (p.isUnpacking())  arr=new foo[n];
      PUParray(p,arr,n); //pup each foo in the array
    }

    ~keepsFoos() {delete[] arr;}
};
```

Array of pointers to classes An array of pointers to classes must handle each element separately, since the PUParray routine does not work with pointers. An "allocate" routine to set up the array could simplify this code. More ambitious is to construct a "smart pointer" class that includes a pup routine.

```
class keepsFooPtrs : public mySuperclass {
private:
    int n;
    foo **arr;/*new'd array of n pointer-to-foos*/
public:
    keepsFooPtrs(int n_) {
      n=n_;
      arr=new foo*[n]; // allocate array
      for (int i=0;i<n;i++) arr[i]=new foo(...); // allocate i'th foo
    }
    keepsFooPtrs() { arr=NULL; }

    void pup(PUP::er &p) {
      mySuperclass::pup(p);
      p|n;//pup the array length n
      if (p.isUnpacking()) arr=new foo*[n]; // allocate array
      for (int i=0;i<n;i++) {
        if (p.isUnpacking()) arr[i]=new foo(...); // allocate i'th foo
        p|*arr[i];  //pup the i'th foo
      }
    }

    ~keepsFooPtrs() {
       for (int i=0;i<n;i++) delete arr[i];
       delete[] arr;
     }
};
```

Note that this will not properly handle the case where some elements of the array are actually subclasses of foo, with virtual methods. The PUP::able framework described in the next section can be helpful in this case.

3.16.4 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;
    }
};
PUPmarshall(keepsParent);
```

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.
```

3.16.5 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.

3.16.6 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.

3.16.7 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.

3.16.8 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.

3.17 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.

A recent change to Charm++ is to also support 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 on networks of workstations. CkPrintf routes all terminal output to the charmrun, which is running on the host computer. So, if a chare on processor 3 makes a call to CkPrintf, that call puts the output in a TCP message and sends it to host computer where it will be displayed. This message passing is 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. [19]

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 the 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.

3.18 initnode and initproc routines

$\boxed{\beta}$

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 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 node and *fooProcInit* and *bar::barProcInit* on every processor before the main computation starts. Initnode calls are always executed before initproc calls.

Note that these routines should only do registration, not computation– use a mainchare instead, which gets executed on only one processor, to begin the computation. Initcall routines are typically used to do special registrations and global variable setup before the computation actually begins.

3.19 Other Calls

The following calls provide information about the machines upon which the parallel program is executing. Processing Element refers to a single CPU. Node refers to a single machine– a set of processing elements which share memory (i.e. an address space). Processing Elements and Nodes are numbered, starting from zero.

Thus if a parallel program is executing on one 4-processor workstation and one 2-processor workstation, there would be 6 processing elements (0, 1 ,2, 3, 4, and 5) but only 2 nodes (0 and 1). A given node's processing elements are numbered sequentially.

int CkNumPes()

returns the total number of processors, across all nodes.

int CkMyPe()

returns the processor number on which the call was made.

[19] Because of communication latencies, the following scenario is actually possible: Chare 1 does a CkPrintf from processor 1, then creates chare 2 on processor 2. After chare 2's creation, it calls CkPrintf, and the message from chare 2 is displayed before the one from chare 1.

int CkMyRank()
returns the rank number of the processor on which the call was made. Processing elements within a node are ranked starting from zero.
int CkMyNode()
returns the address space number (node number) on which the call was made.
int CkNumNodes()
returns the total number of address spaces.
int CkNodeFirst(int node)
returns the processor number of the first processor in this address space.
int CkNodeSize(int node)
returns the number of processors in the address space on which the call was made.
int CkNodeOf(int pe)
returns the node number on which the call was made.
int CkRankOf(int pe)
returns the rank of the given processor within its node.
The following calls provide commonly needed functions.
void CkAbort(const char *message)
Cause the program to abort, printing the given error message. This routine never returns.
void CkExit()
This call informs the Charm kernel that computation on all processors should terminate. After the currently executing entry method completes, no more messages or entry methods will be called on any other processor. This routine never returns.
void CkExitAfterQuiescence()
This call informs the Charm kernel that computation on all processors should terminate as soon as the machine becomes completely idle–that is, after all messages and entry methods are finished. This is the state of quiescence, as described further in Section 3.13. This routine returns immediately.
double CkCpuTimer()
Returns the current value of the system timer in seconds. The system timer is started when the program begins execution. This timer measures process time (user and system).
double CkWallTimer()
Returns the elapsed time in seconds since the program has started from the wall clock timer.
double CkTimer()
This is an alias for either CkWallTimer on dedicated machines (such as ASCI Red) or CkCpuTimer for machines with multiple user processes per CPU (such as workstation cluster.)

3.20 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.

3.20.1 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.

3.20.2 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 CkArrayIndexMax &idx,int onPE,CkArrayID a);
  virtual void ArraySend(int ep,void *m,const CkArrayIndexMax &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.

3.21 Communication Optimizations Framework

The communications framework in Charm++/converse is aimed at optimizing certain communication patterns. Currently the programmer has to specify the communication pattern. The communications library uses the delegation framework. This is done to make access to the communication library transparent to the programmer.

To access the communication framework you first need to create an instance of the Communication Framework which will work with your messages and optimize that pattern. Each communication framework instance needs to know how to manage the messages. Hence a Strategy needs to be initialized and passed to the instance of the Communication Library framework.

Each strategy inherits from the class *Strategy* and implements its entry methods. Each strategy has its own set of options passed to its constructor. There are two types of strategies in our framework.

- Bracketed Strategies. In bracketed strategies each source Chare (which could be an array element or a group) deposits its entries and then the strategy performs the communication optimization. For example the EachToManyMulticastStrategy is a bracketed strategy. For bracketed strategies a beginIteration and an endIteration must be called before and after making the deposits respectively.

- Streaming Strategies. Streaming strategies perform communication optimizations in the background and do not need beginIteration and endIteration to start processing. They usually combine and process the messages after every few ms (a parameter to the strategy) or after certain number of messages have been deposited.

The communications library currently has the following strategies implemented.

- EachToManyMulticastStrategy. EachToManyMulticastStrategy optimizes collective personalized communication and collective multicast or a combination of both. In collective personalized communication operation each array element sends a distinct message to many other array elements. The sending and receiving elements need not belong to the same array. Collective multicast is a special case of collective personalized communication where each array element sends the same message to the destination array elements.

Strategy * strategy = new EachToManyMulticastStrategy(topology, src_array, dest_array);

if the entire arrays are participating in the strategies. If only sections of the source and destination arrays are participating in the collective operation the following constructor could also be used.

EachToManyMulticastStrategy(int topology, CkArrayID srcarray, CkArrayID destarray, int nsrc, CkArrayIndexMax *indices, int ndest, CkArrayIndexMax *destelements);

Setting nsrc to 0 or ndest to 0 would make the strategy use the entire source or destination array respectively.

The header file for this strategy is EachToManyMulticastStrategy.h which needs to be included in the user program.

The constructor also takes an integer parameter *topology*. This tells the strategy how to optimize the collective personalized communication pattern. The options for topology are USE_DIRECT, USE_MESH, USE_HYPERCBE, USE_GRID. USE_DIRECT sends the message directly. USE_MESH imposes a 2d Mesh virtual topology on the processors so now each processor sends messages to its neighbors in its row and column of the mesh which forward the messages to their correct destinations. USE_HYPERCUBE and USE_GRID impose a hypercube and a 3d Grid topologies on the processors. USE_HYPERCUBE will do best for very small messages and small number of processors, 3d has better performance for slightly higher message sizes and then Mesh starts performing best. The programmer is encouraged to try out all the topologies.

To send a multicast message, the multicast message should inherit from CkMcastBaseMsg as mentioned in section 3.8.13 about Array Section Multicast. To send the multicast message create the appropriate array section proxy. On the array section proxy call the appropriate entry method with this message. This will be explained in detail later.

- StreamingStrategy. In this communication optimization strategy each array element sends many small messages to other array elements but these sends are not synchronized and the library periodically collects messages (or collects when a certain number of messages have been deposited) and sends them to their destinations. The period is a parameter to the StreamingStrategy. It needs to be specified in Milli seconds. 10ms is the default

  Strategy * strategy = new StreamingStrategy(period_in_ms, int nmsgs);

  The header file is StreamingStrategy.h.

The following are the steps to use the communication library in a user program.

3.21.1 Compiling User Code

All user programs that use the communcation library should use the linker option -module comlib.For Example
charmc -o pgm pgm.o -module comlib

3.21.2 Calling the Library from User Code

Calls to the communication library have to be made at two places in the user program. They are :

1. main.C

```
//Include the appropriate header file
#include <EachToManyMulticastStrategy.h>

//In main::main()
//Create a Communication Library Instance
ComlibInstanceHandle cinst = CkGetComlibInstance();
//For bracketed strategies this instance should be sent to
```

```
//all source array elements through a broadcast or as a readonly variable

//Create your strategy
Strategy *strategy = new EachToManyStrategy(USE_MESH, srcarray, destarray);
//or
Strategy *strategy = new StreamingStrategy(10,10);

//Register the strategy
cinst.setStrategy(strategy);
```

2. In the array element

```
//Before calling an entry method whose message should go thorough the
//library the proxy has to be delegated. Create a new copy of the
//proxy and delegate it before using it.

CProxy_Hello dproxy = array_proxy;
ComlibDelegateProxy(&dproxy);
cinst.beginIteration();  //Only for bracketed strategies
dproxy[index].entry();
.....
.....
cinst.endIteration();    //Only for bracketed strategies
//Now all calls to proxy will go through the library.
//So non library calls should use another proxy!
//The begin and end calls do not have to be called for the streaming strategy.

//To send a section multicast
CProxySection_Hello dsec_proxy = CProxySection_Hello::ckNew(arrayid, nelements, indices);
ComlibDelegateProxy(&dsec_proxy);
cinst.beginIteration();  //Only for bracketed strategies
.....
dsec_proxy.entry();
.....
cinst.endIteration();    //Only for bracketed strategies
```

NOTE: Any number of communication framework instances can be created. Each of them should have a strategy registered. For bracketed strategies the instance handles should be remembered and beginIteration and endIteration should be called.

4 Inheritance and Templates in Charm++

Charm++ supports inheritance among Charm++ objects such as chares, groups, and messages. This, along with facilities for generic programming using C++ style templates for Charm++ objects, is a major enhancement over the previous versions of Charm++.

4.1 Chare Inheritance

Chare inheritance makes it possible to remotely invoke methods of a base chare from a proxy of a derived chare. Suppose a base chare is of type *BaseChare*, then the derived chare of type *DerivedChare* needs to be declared in the CHARM++ interface file to be explicitly derived from *BaseChare*. Thus, the constructs in the `.ci` file should look like:

```
chare BaseChare {
  entry BaseChare(someMessage *);
  entry void baseMethod(void);
  ...
}
chare DerivedChare : BaseChare {
  entry DerivedChare(otherMessage *);
  entry void derivedMethod(void);
  ...
}
```

Note that the access specifier public is omitted, because CHARM++ interface translator only needs to know about the public inheritance, and thus public is implicit. A Chare can inherit privately from other classes too, but the CHARM++ interface translator does not need to know about it, because it generates support classes (*proxies*) to remotely invoke only public methods.

The class definitions of both these chares should look like:

```
class BaseChare : public Chare {
  // private or protected data
  public:
    BaseChare(someMessage *);
    void baseMethod(void);
};
class DerivedChare : public BaseChare {
  // private or protected data
  public:
    DerivedChare(otherMessage *);
    void derivedMethod(void);
};
```

Now, it is possible to create a derived chare, and invoke methods of base chare from it, or to assign a derived chare proxy to a base chare proxy as shown below:

```
...
otherMessage *msg = new otherMessage();
CProxy_DerivedChare *pd = new CProxy_DerivedChare(msg);
pd->baseMethod();     // OK
pd->derivedMethod();  // OK
...
Cproxy_BaseChare *pb = pd;
pb->baseMethod();    // OK
pb->derivedMethod(); // COMPILE ERROR
```

Note that C++ calls the default constructor of the base class from any constructor for the derived class where base class constructor is not called explicitly. Therefore, one should always provide a default constructor for the base class, or explicitly call another base class constructor.

Multiple inheritance is also allowed for Chares and Groups. Often, one should make each of the base classes inherit "virtually" from Chare or Group, so that a single copy of Chare or Group exists for each multiply derived class.

Entry methods are inherited in the same manner as methods of sequential C++ objects. To make an entry method virtual, just add the keyword virtual to the corresponding chare method– no change is needed in the interface file. Pure virtual entry methods also require no special description in the interface file.

4.2 Inheritance for Messages

Messages cannot inherit from other messages. A message can, however, inherit from a regular C++ class. For example:

```
//In the .ci file:
  message BaseMessage1;
  message BaseMessage2;

//In the .h file:
  class Base {
    // ...
  };
  class BaseMessage1 : public Base, public CMessage_BaseMessage1 {
    // ...
  };
  class BaseMessage2 : public Base, public CMessage_BaseMessage2 {
    // ...
  };
```

Messages cannot contain virtual methods or virtual base classes unless you use a packed message. Parameter marshalling has complete support for inheritance, virtual methods, and virtual base classes via the PUP::able framework.

4.3 Generic Programming Using Templates

One can write "templated" code for Chares, Groups, Messages and other CHARM++ entities using familiar C++ template syntax (almost). The CHARM++ interface translator now recognizes most of the C++ templates syntax, including a variety of formal parameters, default parameters, etc. However, not all C++ compilers currently recognize templates in ANSI drafts, therefore the code generated by CHARM++ for templates may not be acceptable to some current C++ compilers[20].

The CHARM++ interface file should contain the template definitions as well as the instantiation. For example, if a message class *TMessage* is templated with a formal type parameter *DType*, then every instantiation of *TMessage* should be specified in the CHARM++ interface file. An example will illustrate this better:

```
  template <class DType=int, int N=3> message TMessage;
  message TMessage<>; // same as TMessage<int,3>
  message TMessage<double>; // same as TMessage<double, 3>
  message TMessage<UserType, 1>;
```

[20] Most modern C++ compilers belong to one of the two camps. One that supports Borland style template instantiation, and the other that supports AT&T Cfront style template instantiation. In the first, code is generated for the source file where the instantiation is seen. GNU C++ falls in this category. In the second, which template is to be instantiated, and where the templated code is seen is noted in a separate area (typically a local directory), and then just before linking all the template instantiations are generated. Solaris CC 5.0 belongs to this category. For templates to work for compilers in the first category such as for GNU C++ all the templated code needs to be visible to the compiler at the point of instantiation, that is, while compiling the source file containing the template instantiation. For a variety of reasons, CHARM++ interface translator cannot generate all the templated code in the declarations file `*.decl.h`, which is included in the source file where templates are instantiated. Thus, for CHARM++ generated templates to work for GNU C++ even the definitions file `*.def.h` should be included in the C++ source file. However, this file may contain other definitions apart from templates that will be duplicated if the same file is included in more than one source files. To alleviate this problem, we have to do a little trick. Fortunately, this trick works for compilers supporting both Borland-style and Cfront-style template instantiation, therefore, code using this trick will be portable. The trick is to include `*.def.h` with a preprocessor symbol CK_TEMPLATES_ONLY defined, whenever templates defined in an `extern` module are instantiated. If your interface file does not contain template declarations or definitions, you need not bother about including `*.def.h` for `extern` modules. For example, if module `stlib` contains template definitions, that you may want to instantiate in another module called `pgm`, then `pgm.C` should include `stlib.def.h` with CK_TEMPLATES_ONLY defined. Of course, `stlib.decl.h` needs to be included at the top of `pgm.C`.

Note the use of default template parameters. It is not necessary for template definitions and template instantiations to be part of the same module. Thus, templates could be defined in one module, and could be instantiated in another module , as long as the module defining a template is imported into the other module using the extern module construct. Thus it is possible to build a standard CHARM++ template library. Here we give a flavor of possibilities:

```
module SCTL {
  template <class dtype> message  Singleton;
  template <class dtype> group Reducer {
    entry Reducer(void);
    entry void submit(Singleton<dtype> *);
  }
  template <class dtype> chare ReductionClient {
    entry void recvResult(Singleton<dtype> *);
  }
};

module User {
  extern module SCTL;
  message Singleton<int>;
  group Reducer<int>;
  chare RedcutionClient<int>;
  chare UserClient : ReductionClient<int> {
    entry UserClient(void);
  }
};
```

The *Singleton* message is a template for storing one element of any *dtype*. The *Reducer* is a group template for a spanning-tree reduction, which is started by submitting data to the local branch. It also contains a public method to register the *ReductionClient* (or any of its derived types), which acts as a callback to receive results of a reduction.

5 Checkpoint/Restart

CHARM++ offers a range of fault tolerance capabilities through its checkpoint/restart mechanism. Usual Chare array-based CHARM++ application including AMPI application can be checkpointed to disk files and later on restarting from the files.

The basic idea behind this is straightforward: Checkpointing an application is like migrating its parallel objects from the processors onto disks, and restarting is the reverse. Thanks to the migration utilities like PUP'ing(Section 3.16), users can decide what data to save in checkpoints and how to save them.

Two schemes of fault tolerance protocols are implemented.

5.1 Disk-based Checkpoint/Restart

5.1.1 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 holding the intermediate results and 1 holding the final results of each timestep. To save resource, the PUP routine can well omit the 8 intermediate matrices and checkpoint the matrix with final results of each timestep.

Group and nodegroup objects(Section 3.9) are normally not meant to be migrated. In order to checkpoint them, however, the user wants 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 needs mainchares 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. In addition, the programmer also needs to put the proxy to the mainchare (usually noted as mainproxy) as a read-only data in the code, and make sure processor 0, which holds the mainchare, initiates the checkpoint.

After `CkStartCheckpoint` is executed, a directory of the designated name is created and a collection of checkpoint files are written into it.

5.1.2 Restarting

The user can choose to run the Charm++ application in restart mode, i.e., restarting execution from last 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 different number of physical processor. This provides the flexibility to expand or shrink your application when the availability of computing resource changes.

Note that on restart, if the old 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 entry method of a chare object. Since all the entry methods are registered inside Charm++ system, in restart phase, the reduction client will be automatically restored.

After a failure, the system may consist less number of processors. After a problem fixed, some processors may become available again. Therefore, the user may need to flexibility to restart on different number of processors than in the checkpointing phase. This is allowable by giving 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 balancing (See Section 3.11) 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 propagate to all the processors.

5.1.3 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 is 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 routing 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 aree two types of chare arrays, one maintaining intermediate results while the other type holding the final result for each timestep. The programmer can take advantage of the flexibility by omitting PUP'er routine empty for intermediate objects, and do save/restore only for the important objects.

5.2 Double Memory/Disk Checkpoint/Restart

The previous disk-based fault-tolerance scheme is 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 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 storage (as needed in the previous method). Instead, it stores two copies of checkpoint data to two different locations (can be memory or 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 variation stores checkpoints into local disk, thus can be useful for applications with large memory footprint.

5.2.1 Checkpointing

The function that user can call to initiate a checkpointing in a Chare array-based application is:

```
void CkStartMemCheckpoint(CkCallback &cb)
```

where *cb* has the same meaning as in the Section 5.1.1 . Just like the above disk checkpoint described, it is up to programmer what to save. The programmer is responsible for choosing when to activate checkpointing so that the size of a global checkpoint state can be minimal.

In AMPI applications, user just needs to call the following function to start checkpointing:

```
void AMPI_MemCheckpoint()
```

5.2.2 Restarting

When a processor crashes, the restart protocol will be automatically invoked to recover all objects using the last checkpoints. And then the program will continue to run on the survived 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-toerlance 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 crash. In order to achieve this, Charm++ needs to be compiled with the marco option *CK_NO_PROC_POOL* turned on.

5.2.3 Double in-disk checkpoint/restart

A variation of double memory checkpoint/restart, *double in-disk checkpoint/restart*, can be applied to aplcaitions 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]-XXXXXX” and are stored under /tmp.

A programmer can use runtime option *+ftc_disk* to switch to this mode. For example:

```
./charmrun hello +p8 +ftc_disk
```

A Structured Dagger

Charm++ is based on the Message-Driven parallel programming paradigm. The message-driven programming style avoids the use of blocking receives and allows overlap of computation and communication by scheduling computations depending on availability of data. This programing style enables Charm++ programs to tolerate communication latencies adaptively. Threads suffer from loss of performance due to context-switching overheads and limited scalability due to large and unpredictable stack memory requirements, when used in a data-driven manner to coordinate a sequence of remotely triggered actions.

The need to sequence remotely triggered actions arises in many situations. Let us consider an example:

```
class compute_object : public Chare {
private:
int         count;
Patch       *first, *second;
public:
compute_object(MSG *msg) {
count = 2; MyChareID(&chareid);
PatchManager->Get(msg->first_index, recv_first, &thishandle,NOWAIT);
PatchManager->Get(msg->second_index, recv_second, &thishandle,NOWAIT);
}
void recv_first(PATCH_MSG *msg) {
 first = msg->patch;
 filter(first);
 if (--count == 0 ) computeInteractions(first,second);
}
void recv_second(PATCH_MSG *msg){
 second = msg->patch;
 filter(second);
 if (--count == 0) computeInteractions(first,second);
}
}
```

Figure 2: Compute Object in a Molecular Dynamics Application

Consider an algorithm for computing cutoff-based pairwise interactions between atoms in a molecular dynamics application, where interaction between atoms is considered only when they are within some cut-off distance of each other. This algorithm is based on a combination of task and spatial decompositions of the molecular system. The bounding box for the molecule is divided into a number of cubes (*Patches*) each containing some number of atoms. Since each patch contains a different number of atoms and these atoms migrate between patches as simulation progresses, a dynamic load balancing scheme is used. In this scheme, the task of computing the pairwise interactions between atoms of all pairs of patches is divided among a number of *Compute Objects*. These compute objects are assigned at runtime to different processors. The initialization message for each compute object contains the indices of the patches. The patches themselves are distributed across processors. Mapping information of patches to processors is maintained by a replicated object called *PatchManager*. Figure 2 illustrates the CHARM++ implementation of the compute object. Each compute object requests information about both patches assigned to it from the PatchManager. PatchManager then contacts the appropriate processors and delivers the patch information to the requesting compute object. The compute object, after receiving information about each patch, determines which atoms in a patch do not interact with atoms in another patch since they are separated by more than the cut-off distance. This is done in method `filter`. Filtering could be done after both patches arrive. However, in order to increase processor utilization, we do it immediately after any patch arrives. Since the patches can arrive at the requesting compute object in any order, the compute object has to buffer the received patches, and maintain state information using counters or flags. This example has been chosen for simplicity in order to demonstrate the necessity of counters and buffers. In general, a parallel algorithm may have more interactions leading to the use of many counters, flags, and message buffers, which complicates program development significantly.

Threads are typically used to perform the abovementioned sequencing. Lets us code our previous example using threads.

Contrast the compute chare-object example in figure 2 with a thread-based implementation of the same scheme in figure 3. Functions *getFirst*, and *getSecond* send messages asynchronously to the PatchManager, requesting that the specified patches be sent to them, and return immediately. Since these messages with patches could arrive in any order, two threads, *recvFirst* and *recvSecond*, are created. These threads block,

```
void compute_thread(int first_index, int second_index)
{
    getPatch(first_index);
    getPatch(second_index);
    threadId[0] = createThread(recvFirst);
    threadId[1] = createThread(recvSecond);
    threadJoin(2, threadId);
    computeInteractions(first, second);
  }
  void recvFirst(void)
  {
    recv(first, sizeof(Patch), ANY_PE, FIRST_TAG);
    filter(first);
  }
  void recvSecond(void)
  {
    recv(second, sizeof(Patch), ANY_PE, SECOND_TAG);
    filter(second);
  }
```

Figure 3: Compute Thread in a Molecular Dynamics Application

waiting for messages to arrive. After each message arrives, each thread performs the filtering operation. The main thread waits for these two threads to complete, and then computes the pairwise interactions. Though the programming complexity of buffering the messages and maintaining the counters has been eliminated in this implementation, considerable overhead in the form of thread creation, and synchronization in the form of *join* has been added. Let us now code the same example in *Structured Dagger*. It reduces the parallel programming complexity without adding any significant overhead.

```
  array[1D] compute_object {
    entry void recv_first(Patch *first);
    entry void recv_second(Patch *first);
    entry void compute_object(MSG *msg){
      atomic {
         PatchManager->Get(msg->first_index,...);
         PatchManager->Get(msg->second_index,...);
      }
      overlap {
        when recv_first(Patch *first) atomic { filter(first); }
        when recv_second(Patch *second) atomic { filter(second); }
      }
      atomic { computeInteractions(first, second); }
    }
  }
```

Figure 4: *Structured Dagger* Implementation of the Compute Object

Structured Dagger is a coordination language built on top of Charm++ that supports the sequencing mentioned above, while overcoming limitations of thread-based languages, and facilitating a clear expression of flow of control within the object without losing the performance benefits of adaptive message-driven execution. In other words, *Structured Dagger* is a structured notation for specifying intra-process control dependences in message-driven programs. It combines the efficiency of message-driven execution with the explicitness of control specification. *Structured Dagger* allows easy expression of dependences among messages and computations and also among computations within the same object using when-blocks and various structured constructs. *Structured Dagger* is adequate for expressing control-dependencies that form a series-parallel control-flow graph. *Structured Dagger* has been developed on top of CHARM++˙*Structured Dagger* allows CHARM++ entry methods (in chares, groups or arrays) to specify code (a when-block body) to be executed upon occurrence of certain events. These events (or guards of a when-block) are entry methods of the object that can be invoked remotely. While writing a *Structured Dagger* program, one has to declare these entries in CHARM++ interface file. The implementation of the entry methods that contain the when-block is written using the *Structured Dagger* language. Grammar of *Structured Dagger* is given in the EBNF form below.

A.1 Usage

Structured Dagger code can be inserted into the .ci file for any array, group, or chare's entry methods.

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 file).
- Adding a call to the routine "__sdag_init();" from every constructor, including the migration constructor. Forgetting this results in using uninitalized data, and a horrible runtime crash.
- 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(...) {
       __sdag_init();
       ...
    }
    Foo(CkMigrateMessage *m) {
       __sdag_init();
    }

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

For more details regarding *Structured Dagger*, look at the examples located in `pgms/sdag` directory in the CHARM++ distribution.

A.2 Grammar

A.2.1 Tokens

```
<ident> = Valid C++ identifier
<int-expr> = Valid C++ integer expression
<C++-code> = Valid C++ code
```

A.2.2 Grammar in EBNF Form

```
<sdag> := <class-decl> <sdagentry>+

<class-decl> := "class" <ident>

<sdagentry> := "sdagentry" <ident> "(" <ident> "*" <ident> ")" <body>

<body> := <stmt>
        | "{" <stmt>+ "}"

<stmt> := <overlap-stmt>
        | <when-stmt>
        | <atomic-stmt>
        | <if-stmt>
        | <while-stmt>
        | <for-stmt>
        | <forall-stmt>

<overlap-stmt> := "overlap" <body>

<atomic-stmt> := "atomic" "{" <C++-code> "}"

<if-stmt> := "if" "(" <int-expr> ")" <body> [<else-stmt>]

<else-stmt> := "else" <body>

<while-stmt> := "while" "(" <int-expr> ")" <body>

<for-stmt> := "for" "(" <c++-code> ";" <int-expr> ";" <c++-code> ")" <body>

<forall-stmt> := "forall" "[" <ident> "]" "(" <range-stride> ")" <body>

<range-stride> := <int-expr> ":" <int-expr> "," <int-expr>

<when-stmt> := "when" <entry-list>  <body>

<entry-list> := <entry>
              | <entry> [ "," <entry-list> ]

<entry> := <ident> [ "[" <int-expr> "]" ] "(" <ident> "*" <ident> ")"
```

B Further Information

B.1 Related Publications

For starters, see the publications, reports, and manuals on the Parallel Programming Laboratory website: `http://charm.cs.uiuc.edu/`.

B.2 Associated Tools and Libraries

Several tools and libraries are provided for CHARM++. PROJECTIONS is an automatic performance analysis tool which provides the user with information about the parallel behavior of CHARM++ programs. The purpose of implementing Charm++ standard libraries is to reduce the time needed to develop parallel applications with the help of a set of efficient and re-usable modules. Most of the libraries have been described in a separate manual.

B.2.1 Projections

Projections is a performance visualization and feedback tool. The system has a much more refined understanding of user computation than is possible in traditional tools.

Projections displays information about the request for creation and the actual creation of tasks in Charm++ programs. Projections also provides the function of post-mortem clock synchronization. Additionally, it can also automatically partition the execution of the running program into logically separate units, and automatically analyzes each individual partition.

Future versions will be able to provide recommendations/suggestions for improving performance as well.

B.3 Contacts

While we can promise neither bug-free software nor immediate solutions to all problems, Charm++ is a stable system and it is our intention to keep it as up-to-date and usable as our resources will allow by responding quickly to questions and bug reports. To that end, there are mechanisms in place for contacting Charm users and developers.

Our software is made available for research use and evaluation. For the latest software distribution, further information about Converse/Charm++ and information on how to contact the Parallel Programming laboratory, see our website at `http://charm.cs.uiuc.edu/`.

If retrieval of a publication via these channels is not possible, please send electronic mail to `kale@cs.uiuc.edu` or postal mail to:

```
Laxmikant Kale
Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, IL 61801
```