The Charm++ Programming Language Manual

Version 5.0

July 28, 2000

Acknowlegements

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 that, along with the CHARM++ runtime, became the CHARM++ language. The current version (5.0) 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 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. IDL bindings for CHARM++ were developed by Jayant Desouza, Milind Bhandarkar and Gengbin Zheng. 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.

Contents

1 Introduction **5**

1.1 Overview . . . 5

1.2 History . . . 6

1.3 Notation Used . . . 7

2 Charm++ Overview **8**

2.1 Charm++ Execution Model . . . 8

2.2 Entities in Charm++ programs . . . 10

2.2.1 Sequential Objects . . . 10

2.2.2 Messages . . . 10

2.2.3 Chares . . . 10

2.2.4 Chare Groups . . . 11

2.2.5 Chare Nodegroups . . . 11

2.2.6 Chare Arrays . . . 11

3 The Charm++ Language **12**

3.1 Modules . . . 12

3.1.1 Structure of a Charm++ Program . . . 12

3.1.2 Functions in the "decl.h" and "def.h" files . . . 14

3.2 Messages . . . 14

3.2.1 Message Creation and Deletion . . . 15

3.2.2 Message Packing . . . 16

3.2.3 Variable Size Messages . . . 18

3.2.4 Prioritized Execution . . . 20

3.2.5 Entry Method Attributes . . . 23

3.3 Sequential Objects . . . 23

3.4 Chare Objects . . . 23

3.4.1 Chare Creation . . . 24

3.4.2 Method Invocation on Chares . . . 26

3.5 Group Objects . . . 26

3.5.1 Group Creation . . . 27

3.5.2 Method Invocation on Groups . . . 28

3.6 Nodegroup Objects . . . 29
3.6.1 Method Invocation on Nodegroups . . . 30
3.7 Chare Arrays . . . 30
3.7.1 Creating an Array . . . 30
3.7.2 Inserting and Destroying Array Elements . . . 31
3.7.3 Messages and Broadcasts . . . 32
3.7.4 Migratable Array Elements . . . 34
3.7.5 Load Balancing Chare Arrays . . . 36
3.7.6 Reductions on Chare Arrays . . . 36
3.7.7 Built-in Array Reduction Types . . . 37
3.7.8 Defining a new array index type . . . 38
3.7.9 Array Syntax Changes . . . 39
3.8 Read-only Variables, Messages and Arrays . . . 40
3.9 Quiescence Detection . . . 41
3.10 Terminal I/O . . . 41
3.11 Other Calls . . . 42

4 Inheritance and Templates in Charm++ 44
4.1 Chare Inheritance . . . 44
4.2 Inheritance for Messages . . . 45
4.3 Generic Programming Using Templates . . . 45

A Compiling, Running and Debugging Charm++/Converse Programs 47
A.1 Compiling Converse, Charm, and Charm++ Programs . . . 47
A.2 Executing Converse/Charm/Charm++ Programs . . . 48
A.2.1 Command Line Options . . . 50
A.2.2 Additional Uniprocessor Command Line Options . . . 51
A.2.3 Additional Network Command Line Options . . . 51

B CHARM++ Keywords 52

C Syntax Changes from CHARM++ 4.9 55

D Structured Dagger 56
D.1 Grammar . . . 59

D.1.1 Tokens . . . 59
D.1.2 Grammar in EBNF Form . . . 59

E Interface Description Language (IDL) 60

F Further Information 61
F.1 Related Publications . . . 61
F.2 Associated Tools and Libraries . . . 62
F.2.1 Projections . . . 62
F.2.2 TeMPO . . . 62
F.2.3 Communication . . . 64
F.2.4 Barrier . . . 64
F.3 Contacts . . . 66

References 46

Index 47

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 IBM SP, SGI Origin2000, 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), and Intel x86 (Linux, Windows NT). 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.

For a description of the C-based CHARM parallel programming system, please refer to the *Charm Programming Language Manual* and the *Tutorial Introduction to Charm*[2].

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:

[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]CHARM is no longer actively supported and maintained, and these manuals are kept only for offering the historical perspectives.

- Concurrent Objects : called *chares*[3]
- Communication Objects : Messages
- Chare Groups and Nodegroups
- Chare Arrays
- Readonly data

CHARM++ starts a program by creating a single instance of each *mainchare* on processor 0, and invokes constructor methods of these chare. 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.

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 fo 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[4, 5]. The CHARM++ runtime system was retargetted on top of

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

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.

1.3 Notation Used

Small code samples used to illustrate syntax specifications throughout this document will use the following typeface conventions:

- Language keywords appear as boldface words: **chare**.
- User-defined types and function names appear in a sans serif font: chareName.
- User-defined variables appear italicized: *myChare*.
- All other code appears in the same font as the regular text of this document.

Longer code samples of actual code will appear in the standard typewriter font.

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 (or functions, or subroutines, or procedures, or other such thingies) 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. 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 implies 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[4]. Since the actual CHARM++ object on which the method is being invoked is possibly on a remote processor, in possibly different address space, the C++ way of referring to an object, such as a pointer, is not valid in CHARM++. Instead, there are two ways of referring to a remote object in CHARM++. First, every remote object has a unique *handle* called `ChareID`. This is a C struct that uniquely defines the location of the object on the parallel machine. Note that, whatever the type of object (its class), it has the same kind of such handle. The second, and preferred way of referring to a remote object is its "proxy".

Those familiar with various component models in the distributed computing world will recognize "proxy" to be a dummy thingie that refers to the actual thingie. It is exactly that way in CHARM++ too. For each object type (each class), its "proxy" class exists (actually it is generated by the "interface translator"), which contains an actual reference to the class (such as a `ChareID`). Furthermore, the methods of this "proxy" class correspond to the methods of 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. Creation of a proxy results in creation of a new 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 almost always remains the same, and that is of invoking methods on the remote object by invoking methods on proxies.

2.1 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

[4] 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 seemlessly with user-level threads to also support synchronous method execution, albeit with a slight overhead of thread creation and scheduling.

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. It will not process two messages for the same chare concurrently, but otherwise it is free to schedule them in any way.

Methods of a chare that can be remotely invoked are called *entry* methods. Entry methods may take a pointer to a message object, or no parameters. Since chares can be created on remote processors, obviously some constructor of a chare needs to be an entry method.

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

Other CHARM++ objects are collections of chares. They are: *chare-groups*, *chare-nodegroups*, and *chare-arrays*, referred to as *groups*, *nodegroups*, and *arrays* throughout this manual. A group (nodegroup) is a collection of chares, one per processor (SMP node), that is addressed using a unique system-wide name. An array is a collection of arbitrary number of migratable chares, mapped to processors according to a user-defined map group.

Every CHARM++ program must have at least one **mainchare**. There can be only one instance of this , which is created on processor 0 when the CHARM++ program starts up. Execution of a CHARM++ program begins with Charm Kernel constructing all the designated **mainchare**s. Typically, the **mainchare** constructor starts the computation by creating other chares and chare 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 about 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-defind entities, and maintaining these identifiers can be cumbersome. Fortunately, it can be done 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. This call is not the same as **exit()** in Unix: it merely informs the Charm Kernel that all computations on all processors must be terminated, and then returns to the user program: the programmer should make sure that no work is done by the method after **CkExit()** is called. Charm Kernel ensures that no messages are processed and no entry methods are called after the currently executing entry method completes. **CkExit()** need not be called on all processors; it is enough to call it from just one processor at the end of the computation.

2.2 Entities in CHARM++ programs

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

2.2.1 Sequential Objects

A CHARM++ program typically contains a number of sequential objects. These objects are similar to the objects in C++, except that they *should not have any static data members*[5]. These objects are created synchronously on the local processor (using a `new` operator in C++), and their methods could be synchronously invoked only from the local processor. These objects are not known to the CHARM++ runtime system, and thus they need not be mentioned in the module interface files.

2.2.2 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 should be specified in the interface file of the module. 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 methods: `pack`, and `unpack`. These messages are specified differently in the interface file for the module. The third variation of communication objects is called *varsize* messages. Varsize messages is an effective optimization on conditionally packed messages, and have to be made known to the CHARM++ runtime using a special keyword in the interface file.

2.2.3 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 constuctors as entry methods (with at most one message pointer parameter). These chares and their entry methods have to be specified in the interface file.

[5]This restriction makes CHARM++ programs portable across distributed and shared memory architectures, as well as clusters of shared memory multiprocessors. If one feels that these restrictions are too severe, one is encouraged to look at the portability macros of Converse, that make it possible to write portable programs in presence of global or static variables.

2.2.4 Chare Groups

Chare Groups[6] 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.

2.2.5 Chare Nodegroups

Chare nodegroups are very similar to chare groups except that instead of having one groupmember 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. Sematically, methods invoked on a nodegroup member could be executed on any processor within that node. This fact can be 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.

2.2.6 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. However, what distinguishes chare arrays from the other entities in CHARM++ is that the dynamic load balancing framework in CHARM++ (LB Framework) treats them 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 some smart strategies to redistribute array elements across the available processors. Chare arrays is a recent addition to CHARM++ in response to various dynamic and irregular applications of parallel computing. Currently, chare arrays are implemented as a library on top of the Charm kernel. However, the CHARM++ interface translator recognizes them and has special syntax for describing them.

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

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 [7]. 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 modules 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[8]. 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) [9].

A simple Charm++ program is given below:

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

module Hello {
```

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

[8] 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.

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

```
  readonly mid;
  mainchare HelloMain {
    entry HelloMain(); // implicit CkArgMsg * as argument
    entry void Wait(void);
  };
  group HelloGroup {
    entry HelloGroup(void);
  }
};

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

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

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

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

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

void HelloMain::Wait(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;
  CProxy_HelloMain pself(mid);
  pself.Wait();
}

#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 Wait method of HelloMain. The Wait 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 Messages

In CHARM++, chares, groups and nodegroups communicate using messages. Sending a message to an object corresponds to an asynchronous method invocation. Several variations on messaging are available, regular CHARM++messages are objects of *fixed size*. If you want your message object to contain pointers to dynamically allocated memory and need these to be valid when messages are sent across processors you must declare your messages to be **varsize** (variable size) or **packed** messages, which need some additional processing. Though similar in nature, they vary in the amount of flexibilty they provide in terms of representing arbitrary data structures.

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.

Message definitions that may appear in the `.h` or `.C` file are very similar to class definitions in C++.

In the `.ci` file (the interface file), a message is declared as:

message MessageType;

a variable sized or packed message is declared as:

message [varsize—packed] MessageType;

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++
};

3.2.1 Message Creation and Deletion

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

MessageType *$msgptr$ =
    new [((**int** *)*sizes_array*, **int** *priobits*=0)] MessageType[(constructor arguments)];

The optional arguments to the new operator are used when allocating **varsize** or **prioritized** messages. The *sizes_array* used for varsize messages is an array of integer elements of which 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 that will be used to store the message priority.

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

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

assign the size array and allocate the message:

```
int size[2];
size[0] = 10;
size[1] = 20;
VarsizeMessage *msg = new (size,0) VarsizeMessage;
```

This allocates a *VarsizeMessage*, in which (typically) *firstArray* points to an array of 10 ints and *secondArray* points to an array of 20 doubles. We say typically because the programmer has full control over how the sizes array is used. The meaning of this will be made clear in the section on variable sized messages.

To add a priority bitfield to this message,

```
VarsizeMessage *msg = new (size, 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 3.2.2 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.

3.2.2 Message Packing

When one declares CHARM++ messages in the ".ci" file, the translator generates code to register those messages along with their size with the runtime system. This information is later used to efficiently allocate messages by the runtime system.

In many cases, the messages store *non-linear* data structures 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. This is provided in CHARM++ using **packed** messages.

Such messages are declared in the `.ci` file as:

```
message [packed] PMessage;
```

The class PMessage needs to inherit from CMessage_PMessage and needs to 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
module PackExample {
  ...
  message [packed] PackedMessage;
  ...
};

// File: pgm.h
...
class PackedMessage : public CMessage_PackedMessage
{
  public:
    BinaryTree<NodeType> *btree; // Non-linear data structure used in e.g. only
    static void *pack(PackedMessage *);
    static PackedMessage *unpack(void *);
    ...
};
...

// File: pgm.C
...
void *
PackedMessage::pack(PackedMessage *inmsg)
{
  int numnodes = inmsg->btree->numNodes();
  int totalsize = numnodes*sizeof(NodeType) + sizeof(int);
  int *buf = (int *)CkAllocBuffer(inmsg, totalsize);
  // buf is now just raw memory to store the data structure
  *buf = numnodes;           // copy numnodes into buffer
  buf++;      // don't overwrite numnodes
  inmsg->btree->flatten((void *)buf); // copies nodes, and
                                      // replaces pointers by indices
  buf--;                     // don't lose numnodes
```

```
  delete inmsg;
  return (void *)(buf);
}

PackedMessage *
PackedMessage::unpack(void *inbuf)
{
  // inbuf is the raw memory allocated and assigned in pack
  int *buf = (int *)inbuf;  // stop the compiler from whining
  int numnodes = *buf;      // pull out numnodes from the buffer
  buf++;
   // allocate the message through the charm kernel
  PackedMessage *pmsg = (PackedMessage*)CkAllocBuffer(inbuf, sizeof(PackedMessage));
   // call the Constructor of PackedMessage that takes a raw buffer in
   //   the format numnodes, flattened btree and constructs itself
  pmsg = new ((void *)pmsg) PackedMessage(inbuf, numnodes); // constructs btree
  CkFreeMsg(inbuf);
  return pmsg;
}
...
PackedMessage *pm = new PackedMessage;  // just like always
pm->bTree = new BinaryTree<NodeType>;
...
```

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.

3.2.3 Variable Size Messages

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.

A **varsize** message is declared as `message [varsize] mtype` in CHARM++ interface file. The class mtype has to inherit from CMessage_mtype AND has to provide **pack**, **unpack**, AND **alloc** class methods.

Thus, a varsize message is declared in the interface as:

```
message [varsize] MyVarsizeMessage;
```

In the C++ header file, it has to be declared as:

```
class MyVarsizeMessage : public CMessage_MyVarsizeMessage {
// Other methods & data members
  static void *alloc(int msgnum, size_t size, int *array, int priobits);
  static void* pack(MyVarsizeMessage* in);
  static MyVarSizeMessage* unpack(void* in);
```

```
};
```

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);
```

An Example:

Suppose a CHARM++ message contains two varsize fields:

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

Then in the `.ci` file, this can be represented as:

```
message [varsize] NewVarsizeMessage;
```

and in the `.h` or `.C` file as:

```
class NewVarsizeMessage : public CMessage_NewVarsizeMessage {
  public:
    int len;
    int *firstArray;
    double *secondArray;

  static void *alloc(int msgnum, int size, int *array, int priobits)
  {
    int totalsize;
    // 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 size/totalsize is not
    //   altered with ALIGN8.
    size = ALIGN8(size);       // 8-byte align size with this macro
    totalsize = size + array[0]*sizeof(int) + array[1]*sizeof(double);
    NewVarsizeMessage *newMsg =
      (NewVarsizeMessage *) CkAllocMsg(msgnum, totalsize, priobits);

    // make firstArray point to the end of newMsg in memory
```

```
    newMsg->firstArray = (int *) ((char *)newMsg+size);
    // make secondArray point to after the end of first array
    newMsg->secondArray = (double *) ((char *)newMsg + size +
                                      sizeof(int)*array[0]);
    return (void *)newMsg;
  }

  static void *pack(NewVarsizeMessage *in)
  {
    // set firstArray an offset from the start of in
    in->firstArray = (int *) ((char *) in->firstArray -
                              (char *)&in->firstArray);
    // set secondArray to the appropriate offset
    in->secondArray = (double *) ((char *) in->secondArray -
                                  (char *) &in->secondArray);
    return in;
  }

  static NewVarsizeMessage* unpack(void *in)
  {
    NewVarsizeMessage *me = (NewVarsizeMessage *) in;  // stop compiler whining

    // return firstArray to an absolute address in memory
    me->firstArray = (int *) ((char *) &(me->firstArray) +
                              (size_t)(me->firstArray));
    // likewise for secondArray
    me->secondArray = (double *) ((char *) &(me->secondArray) +
                                  (size_t)(me->secondArray));
    return me;
  }
};

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

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.

3.2.4 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 3.2.1. 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.

3.2.5 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(MessageType *);
```

Charm++ currently offers four attributes that one may give an entry method: threaded, sync, exclusive, and virtual.

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.

Virtual entry methods are inherited in the same manner as virtual methods of sequential C++ objects. For a detailed discussion of inheritance in Charm++, refer to section 4. To make an entry method virtual, just add the keyword **virtual** to that method's attribute list. Additionally, one may make a virtual entry method a pure virtual entry method. They behave in the same way as pure virtual methods in C++ and are declared in a very similar fashion. To make a virtual entry method pure, add a "= 0" after that entry method's .ci file declaration. This looks like the following:

```
entry [virtual] void PureVirtualEntry(MessageType *) = 0;
```

3.3 Sequential Objects

These are the same as C++ classes and functions. All C++ features can be used. However, care needs to be taken when sequential objects interact with CHARM++ objects.

3.4 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(MessageType1 *);
    entry void EntryMethodName2(MessageType2 *);
};
```

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 method* definitions of the form:
public:
    ChareType(MessageType1 *$MsgPointer$)
        { // C++ code block }
    void EntryMethodName2(MessageType2 *$MsgPointer$)
        { // 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[10]. 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[11].

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.

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.

3.4.1 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(M1 *);
    entry void someEntry(M2 *);
  };
}
```

and in the `mod.h` file:

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

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

```
#include "mod.decl.h"
class C : public Chare {
  public:
    C(M1 *);
    void someEntry(M2 *);
};
```

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(MessageType *$msgPtr$, **CkChareID** *$vHdl$, **int** $destPE$);

where each parameter above is optional and where

- chareType is the name of the type of chare to be created.
- *msgPtr* is a pointer to a message whose type must correspond to the parameter for the constructor entry method chareType(MessageType *) in chare chareType. This parameter may be omitted if the constructor takes a void parameter.
- *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:

   MessageType *$MsgPtr$ = **new** MessageType;
   CProxy_C *pC = **new** **CProxy_C**($MsgPtr$);

// or
MessageType $*MsgPtr$ = **new** MessageType;
CProxy_C::**ckNew**(*MsgPtr*);

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

   MessageType $*MsgPtr$ = **new** MessageType;
   CProxy_C::**ckNew**(*MsgPtr*, *destPE*);

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

   MessageType $*MsgPtr$ = **new** MessageType;
   CkChareID *cid*;
   CProxy_C::**ckNew**(*MsgPtr*, &*cid*, *destPE*);
   CProxy_C *c*(*cid*);

3.4.2 Method Invocation on Chares

Before sending a message to a chare via an entry method, we need to get a *proxy* of that chare using either the chare identifier, or by creating the proxy directly from the chare creation (see Section 3.4.1. The following code creates a proxy from a **CkChareID**.

CProxy_chareType *chareProxy*(*chareID*);

This declares a proxy named *chareProxy* of chareType using *chareID*. Alternatively, we can create a proxy pointer:

CProxy_chareType $*chareProxyPointer$ = **new** **CProxy_**chareType(*chareID*);

This creates a pointer named *chareProxyPointer* which points to the proxy.

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

chareProxy.EntryMethod(*MessagePointer*)
or,
chareProxyPointer−>EntryMethod(*MessagePointer*)

This sends the message pointed to by *MessagePointer* to the chare whose proxy/proxy pointer is *chareProxy*/*chareProxyPointer* at the entry method EntryMethod, which must be a valid entry method of that chare type. This call is asynchronous and non-blocking; it returns immediately after sending the message.

3.5 Group Objects

A *group*[12] 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

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

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.

3.5.1 Group Creation

Chare Group Declaration:

Given a `.ci` file as follows:

```
group G {
  entry G(M1 *);
  entry void someEntry(M2 *);
};
```

and the following `.h` file:

```
class G : public Group {
  public:
    G(M1 *);
    void someEntry(M2 *);
};
```

we can create a group in a manner similar to a regular chare. Note the difference in how the virtual handle is created.

```
M1 *m1 = new M1;
CProxy_G *pG = new CProxy_G(m1);
  // or
CkGroupID gid = CProxy_G::ckNew(m1);
CProxy_G g(gid);
```

3.5.2 Method Invocation on Groups

Before sending a message to a group via an entry method, we need to get a proxy of that group using the group identifier (a **CkGroupID**). The syntax for obtaining the proxy or a proxy pointer is:

CProxy_groupType *groupProxy*(*groupID*);
or,
CProxy_groupType **groupProxyPointer* = **new CProxy_groupType**(*groupID*);

The first approach creates a proxy to the group represented by *groupID* while the second creates a pointer named *groupProxyPointer* to a proxy to the group represented by *groupID*.

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

groupProxy.EntryMethod(*MessagePointer*, *Processor*)
or,
groupProxyPointer−>EntryMethod(*MessagePointer*, *Processor*)

This sends the message in *MessagePointer* to the branch of the group represented by *groupID* 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 branched chare (i.e., to all processors) using the notation :

groupProxy.EntryMethod(*MessagePointer*)
groupProxyPointer−>EntryMethod(*MessagePointer*)

This sends the message in *MessagePointer* 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.

Note that the programmer relinquishes control of a message after sending it. Further access to the message field can cause runtime errors.

Sequential objects, chares and other groups can access public members of the branch of a group *on their processor* using the following notation:

((GroupType*)**CkLocalBranch**(**CkGroupID** *groupID*))->*DataMember*, and
((GroupType*)**CkLocalBranch**(**CkGroupID** *groupID*))->method().

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. Once a proxy to the local branch of a group is obtained, that branch can be thought of as a regular object. Its public methods can return values, and its public data is readily accessible.

CkLocalBranch returns a generic (`void *`) pointer. It needs to be cast to a pointer of appropriate classe before invoking methods or accessing data members. CHARM++ provides another way to do this using the generated *proxy* classes. One may call the static method **ckLocalBranch** of the proxy class of appropriate group to get the correct type of pointer. For example, method foo can to be invoked on the local branch of a group G with gid as CkGroupID as:

(**CProxy_**G::**ckLocalBranch**(*gid*))->foo(...);

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.

3.6 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 3.2.5.

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 $O(\#$ 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 $O(\#$ 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.

3.6.1 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.

3.7 Chare Arrays

Chare arrays are arbitrarily-sized collections of chares. The entire array is 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 multidimentional array), or an arbitrary string of bytes (e.g., a binary tree index).

In most respects, array elements are identical to chares. Differences include variations in the creation and remote invocation syntax, to include the array index.

Array elements can be migrated, allowing arrays to be efficiently load balanced. Array elements can also receive array broadcasts and contribute to array reductions.

3.7.1 Creating an Array

You can declare a one-dimentional chare array as:

```
//In the .ci file:
```

```
array [1D] A {
  entry A();
  entry void someEntry(M1 *);
};

//In the .h file:
class M1:public CMessage_M1 {
//...message fields...
};

class A : public ArrayElement1D {
  public:
    A(void);
    A(ArrayElementMigrateMessage *);

    void someEntry(M1 *);
};
```

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

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

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

3.7.2 Inserting and Destroying Array Elements

You create an array using the **CProxy_Array::ckNew** routine. You insert elements into the array with the **CProxy_Array::insert** call, and start using the array with **CProxy_Array::doneInserting**.

To create a 1D array containing elements indexed (0, 1, ..., num_elements-1), you can use the shorter:

```
CProxy_A a = CProxy_A::ckNew(num_elements);
```

If A has a constructor that takes a message, you can use

```
CProxy_A a = CProxy_A::ckNew(msg,num_elements);
```

If you don't have a 1D array or don't want a contiguous range of elements, you insert array elements dynamically using the **insert** and **doneInserting** methods:

```
CProxy_A a = CProxy_A::ckNew();//<- note no arguments
a[17].insert();//<- could also pass a message to constructor
a[43].insert();//<- could use a different message or constructor here
a.doneInserting(); //<- needed because we used insert.
```

Insertion allows you to create sparse or multidimentional arrays, send a different message to each element, and specify the PE to create the element on.

The **doneInserting** call starts the the reduction manager (see "Array Reductions") and load balancer (see "Load Balancer")– since these objects need to know about all the array's elements, they must be started after the initial elements are inserted. However, if you forget to call **doneInserting**, the reduction manager and load balancer will never start!

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.

To summarize, if your constructor takes no message, you can use **ckNew** and **insert** as:

```
With 1D elements: ckNew(numInitial); //<- doesn't need doneInserting
No elements:      ckNew(); //<- will need doneInserting
Default PE:       insert();
Specific PE:      insert(onPE);
```

And if your constructor takes a message:

```
With 1D elements: ckNew(msg, numInitial); //<- doesn't need doneInserting
No elements:      ckNew(); //<- will need doneInserting
Default PE:       insert(msg);
Specific PE:      insert(msg,onPE);
```

The system chooses the default PE to create an array element on based on the element's index: 1D indices map to processors in a round-robin fashion; other indices map based on a hash function applied to the index.

To destroy an array element– detach it from the array call its destructor, and release its memory– invoke its "destroy" entry point, as:

```
a3(x,y,z).destroy();
```

After destroying an element, you may insert a new element at its index.

3.7.3 Messages and Broadcasts

Charm++contains direct support for 1D, 2D, 3D, and user-defined index arrays. These arrays can be declared as:

```
//In the .ci file:
message MyMsg;
array [1D] A1 {entry A1();entry void e(MyMsg *m);}
```

```
array [2D] A2 {entry A2();entry void e(MyMsg *m);}
array [3D] A3 {entry A3();entry void e(MyMsg *m);}
array [Foo] AF {entry AF();entry void e(MyMsg *m);}
```

The last declaration expects an array index of type CkArrayIndexFoo, which must be defined before including the .decl.h file (see "Defining a new array index type" below). Also, ¡charm++.h¿ must be included before the definition of the CkArrayIndexFoo type.

```
//In the .h file:
class MyMsg: public CMessage_MyMsg {
public:
  int val;
  MyMsg(int i) {val=i;}
}
class A1:public ArrayElement1D {public: A1(){} ...};
class A2:public ArrayElement2D {public: A2(){} ...};
class A3:public ArrayElement3D {public: A3(){} ...};
class AF:public ArrayElement {public: AF(){} ...};
//(the migration constructors have been omitted for clarity)
```

You can insert elements or invoke methods on the elements of these arrays by specifying the array index in parentheses. For 1D arrays or user-defined arrays, you can also use brackets.

```
//In the .C file:
int x,y,z;
CProxy_A1 a1=CProxy_A1::ckNew();
a1[x  ].insert();
a1(x+1).insert();
a1.doneInserting();
a1[x].e(new MyMsg(7));

CProxy_A2 a2=CProxy_A2::ckNew();
a2(x,y).insert();
a2.doneInserting();
a2(x,y).e(new MyMsg(7));

CProxy_A3 a3=CProxy_A3::ckNew();
a3(x,y,z).insert();
a3.doneInserting();
a3(x,y,z).e(new MyMsg(7);

CProxy_AF aF=CProxy_AF::ckNew();
aF(new CkArrayIndexFoo(...)).insert();
aF[new CkArrayIndexFoo(...)].insert();
aF.doneInserting();
aF[new CkArrayIndexFoo(...)].e(new MyMsg(7);
```

With user-defined index types, you must pass a fresh index (allocated with new) to each [] or () call– the system will delete the index.

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

```
a3.e(new MyMsg(7)); //<- invokes e(7) on each array element
```

3.7.4 Migratable Array Elements

Array objects can migrate from one PE to another. For example, the load balancer (see section "Load Balancing Chare Arrays") 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);//<- call superclass's pup routine
    p(nt);
    p(chr);
    p(flt,7);
    p(numDbl);
    if (p.isUnpacking()) dbl=new double[numDbl];
    p(dbl,numDbl);
}
```

You can apply a **PUP::er** to any primitive variable (as with "p(nt)") or array of primitive variables– there, the second parameter gives the length of the array (as with "p(flt,7)"). You can pass a simple run of bytes by calling the **PUP::er** with a "void *" and number of bytes (packing bytes, however, prevents cross-platform pack/unpacking).

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

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

- 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.

Migration constructors, then, are normally empty– all the unpacking of the data items is done in the element's **pup** routine.

The **PUP::er** overhead is very small– one virtual function call for each item to be packed/unpacked. The actual packing/unpacking is a simple memory-to-memory binary copy. In addition, the **pup** routine may also be called to pack your array element to disk (e.g., for a "checkpoint" operation), and unpack it from disk afterwards (e.g., for "restart" operation). During unpacking, a "translating" **PUP::er** can be used to perform binary format conversion if needed.

To easily pack user-defined types, we recommend adding a **pup** method to the user-defined type:

```
class myData
{
private:
    int a,b;
    char c;
public:
    ...
    void pup(PUP::er &p) {
        p(a);
        p(b);
        p(c);
    }
};

//... array element can now easily pack myData as:
class myElement:public ArrayElement1D
{
private:
    myData d;
public:
    ...
    void pup(PUP::er &p)
    {
        ArrayElement1D::pup(p);//Call superclass
        d.pup(p); //<- pack/unpacks myData
    }
};
```

3.7.5 Load Balancing Chare Arrays

CHARM++includes a run-time load balancer which works on array elements. The idea is to run programs more efficiently by migrating array elements away from overloaded processors. To use the load balancer, you must make your elements migratable (see migration section above) and chose a load balancing "strategy" (see the load balancing section for a description of load balancing strategy routines).

There are two ways to use the load balancer– the AtSync method, for elements that only want to be migrated at certain times; and the non-AtSync method, for elements that can migrate any time.

The non-AtSync method is the default. Your elements may be asked to migrate any time they are not currently executing an entry method.

For the AtSync method, set **usesAtSync** to true in your array element constructor. When an element is ready to migrate, call **ArrayElement::AtSync**. Once all elements have reached AtSync, the load balancer runs and may migrate elements. Once all migrations are complete, the load balancer calls **ArrayElement::ResumeFromSync**. You must ensure no messages are sent to an array element between its calls to AtSync and ResumeFromSync. AtSync/ResumeFromSync is currently a implemented as a global barrier.

3.7.6 Reductions on Chare Arrays

A reduction applies a single operation (e.g. add, max, min, ...) to data items scattered across many processors and collects the result in one place. CHARM++supports reductions 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,void *data,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 array elements must call contribute, and all must use the same reduction type. When you create a new array element, it is expected to contribute to the next reduction not already in progress. The reduction will complete properly even if array elements are migrated or deleted.

If we want to sum each array element's single integer myInt, we would use:

```
//Inside any array element method
int myInt=get_myInt();
contribute(sizeof(int),(void *)&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 all array elements, from each array element call:

```
//Inside any array element method
double forces[2]=get_my_forces();
contribute(2*sizeof(double),(void *)forces,CkReduction::sum_double);
```

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

After the data is reduced, it is passed to a "client" function on node zero. This is an ordinary C or C++ function that you pass to **CProxy_Array::setReductionClient**. Reduction clients are typically registered from your main Chare's constructor, as in:

```
//Outside any class definition:
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;
}

...
    //Inside main::main
    CProxy_A a=CProxy_A::ckNew();
    a.setReductionClient(myClient,(void *)NULL);
```

This simple client prints out the double values received, and does not use the **param** value passed in (**NULL**, in this case). The handler can use param, dataSize, and the data in any fashion; but the data will be deleted after the handler returns.

3.7.7 Built-in Array Reduction Types

CHARM++includes several built-in reduction types, used to combine the contributions from different array elements. Any of them may be passed as an **ArrayReductionFn** type to **ArrayElement::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 single numbers (pass a pointer) or arrays– just pass the correct number of bytes to **ArrayElement**::**contribute**.

1. **CkReduction::sum_int**, **_float**, **_double**– the result will be the sum of the given numbers.
2. **CkReduction::product_int**, **_float**, **_double**– the result will be the product of the given numbers.
3. **CkReduction::max_int**, **_float**, **_double**– the result will be the largest of the given numbers.
4. **CkReduction::min_int**, **_float**, **_double**– the result will be the smallest of the given numbers.
5. **CkReduction::and**– the result will be the logical AND of the given integers. 0 is false, nonzero is true.
6. **CkReduction::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.

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.

3.7.8 Defining a new array index type

CHARM++array indices are arbitrary collections of bytes. To define a new array index, you create an ordinary C++ class which inherits from **CkArrayIndex** and overrides the **getKey** method. 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:

```
class CkArrayIndexFoo:public CkArrayIndex {
public:
    foo f;

    CkArrayIndexFoo(const foo &in) {f=in;}

    //This method returns the length of and a pointer to the index data.
    //The returned pointer must be aligned to at least an integer boundary.
    virtual const unsigned char *getKey(/*out*/ int &nBytes) const {
       nBytes=sizeof(foo);
        return (const unsigned char *)&f;
    }
};
```

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 ArrayElement {public: AF() {} ... }

//in the .C file:
    foo f;
    CProxy_AF a=CProxy_AF::ckNew();
    a[new CkArrayIndexFoo(f)].insert();
    ...
```

Inside an **ArrayElement**, you can get the bytes of your index from the “thisindex” variable, which has type **CkArrayIndexGeneric***. It may be more convenient, however, to inherit from **ArrayElementT¡foo¿**, which has a “thisIndex” (note the capitalization difference) variable of type foo. In this case, you can do:

```
//.ci file unchanged
//in the .h file:
class AF:public ArrayElementT<foo> {public: AF() {} ... }

//in the .C file:
AF::AF()
{
    foo &myF=thisIndex;
    functionTakingAfoo(myF);
}
```

3.7.9 Array Syntax Changes

The syntax for arrays changed in April, 2000 in the following ways:

- CkAID is obsolete, and replaced with CkArrayID. In general, you should now use CProxy’s instead of CkArrayID’s because the syntax is more palatable and the compiler can do static type checking.
- The translator syntax for arrays now includes the index type, e.g. “array [1D] foo ... ”.
- Messages sent to array elements need no longer inherit from ArrayMessage.
- Array elements for 1D arrays must inherit from ArrayElement1D to use a 1D thisIndex. You may still inherit from ArrayElement, but you won’t have a thisIndex.
- You may have multiple array constructors; each can now take a user-defined message, or no message.
- The “ArrayElementCreateMessage” constructor is obsolete; as are the finishConstruction and finishMigration calls. No constructor need explicitly call its superclass.

- For migration, the old packsize, pack, and unpacking portion of the migration constructor have been replaced by a unified pack/unpack "pup" routine.
- For load balancing, the uses_at_sync variable is now called usesAtSync.
- CkReduction_sum_int is now an enumeration, not a function pointer, and now called CkReduction::sum_int.
- CkRegisterArrayReductionHandler(proxy,fn,param) is now proxy.setReductionClient(fn,param).

3.8 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. 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 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++ compiler cannot prevent assignments to read-only variables. The user must make sure that no assignments occur in the program.

3.9 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** takes two parameters: 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*, **CkChareID*** *chareID*);

To retrieve the corresponding index of a particular entry method, you must use a static method contained within the **CProxy** object corresponding to the chare containing that entry method. The syntax of this call is as follows:

CProxy_ChareName::**ckIdx**_EntryMethod(Msg **Message*);

where *chareID* is the name of the chare identifier of the chare containing the desired entry method, EntryMethod is the name of that entry method, and *Message* is a pointer to the kind of message that the desired entry method takes as a parameter. To make this look a little cleaner, we have provided a simple macro called **EntryIndex**, which can be used in the place of this convoluted looking static method call. **EntryIndex** takes as parameters the type of chare in which the entry method is located, the name of the entry method itself, and the type of message that the entry method takes as a parameter. For example:

EntryIndex(ChareName, EntryName, MsgName);

Note that ChareName, EntryName, and MsgName are **NOT** variables or constants. This is text that the preprocessor uses to fill in portions of the previously mentioned static method call. Additionally, this macro method will not work with templated chares (refer to Section 4 for details on templated chares).

CkWaitQd, however, 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.

3.10 Terminal I/O

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

In the 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.

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 processor 0, which is the one running the host program. So, if a chare on processor 3 makes a call to CkPrintf, that call puts the output in a message and sends it to processor 0 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. [13]

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

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 **ckin**, **ckout**, and **ckerr** in the place of cin, 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. **ckout** and **ckerr** both have the same idiosyncratic behavior as CkPrintf, and **ckin** behaves in the same way as CkScanf.

3.11 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.

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()

[13] 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.

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.

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. **CkExit()** should be the last statement of the entry method from which it was called.

double CkTimer()
returns the current value of the system timer in milliseconds. The system timer is started when the program begins execution. This timer measures process time (user and system).

double CkWallTimer()
returns the elapsed time since the program has started from the wall clock timer.

4 Inheritance and Templates in Charm++

CHARM++ 5.0 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 version 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 {
  BaseChare(someMessage *);
  baseMethod(void);
  ...
}
chare DerivedChare : BaseChare {
  DerivedChare(otherMessage *);
  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 *);
    baseMethod(void);
};
class DerivedChare : public BaseChare {
  // private or protected data
  public:
    DerivedChare(otherMessage *);
    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(); // 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 to avoid explicitly calling another base class constructor.

Multiple inheritance is also allowed for Chares and Groups. However, one should make sure that 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.

One may have *virtual* entry methods in base chare classes. They are specified by the entry attribute **virtual** in the interface file. Also, such a method could be pure virtual (where implementation of this method is not given.) It is noted with the usual C++ syntax of "`=0`" even in the interface file. Base classes with pure virtual methods are called *abstract* base classes in OO-lingo. Note that, though instances of abstract base chares cannot be created, a proxy objects of such a type can be created by constructing it with the chareID of the concrete derived chare. Invoking methods on this proxy object would invoke those methods remotely on the concrete derived class instance.

4.2 Inheritance for Messages

Similar to Chares, messages can also be derived from base messages. One needs to specify this in the CHARM++ interface file similar to the Chare inheritance specification (that is, without the **public** access specifier.) Message inheritance makes it possible to send a message of derived type to the method expecting a base class message.

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[14].

[14]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

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>;
```

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.

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`.

A Compiling, Running and Debugging Charm++/Converse Programs

In this section, we give a brief description on how to compile, run and debug a charm++/Converse programs. This is a quick start introduction, only most commonly used options will be mentioned. For more information, refer to Charm/Charm++/Converse Installation and Usage Manual.

A.1 Compiling Converse, Charm, and Charm++ Programs

The `charmc` program standardizes compiling and linking procedures among various machines and operating systems.

Charmc can perform the following tasks. The (simplified) syntax for each of these modes is shown. The options are described next.

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

The following command-line options are most useful to users of charmc:

`-o` *output-file*: Output file name. Note: charmc only ever produces one output file at a time. Because of this, you cannot compile multiple source files at once, unless you then link or archive them into a single output-file. If exactly one source-file is specified, then an output file will be selected by default using the obvious rule (eg, if the input file if pgm.c, the output file is pgm.o). If multiple input files are specified, you must manually specify the name of the output file, which must be a library or executable.

`-c`: Ignored. There for compatibility with `cc`.

`-D*`: Defines preprocessor variables from the command line at compile time.

`-I`: Add a directory to the search path for preprocessor include files.

`-g`: Causes compiled files to include debugging information.

`-L*`: Add a directory to the search path for libraries selected by the `-l` command.

`-l*`: Specifies libraries to link in.

`-O`: Causes files to be compiled with maximum optimization.

`-NO`: If this follows -O on the command line, it turns optimization back off. This is just a convenience for simple-minded makefiles.

`-verbose`: All commands executed by charmc are echoed to stdout.

`-language` {`converse|charm++|sdag|idl`}: When linking with charmc, one must specify the "language". This is just a way to help charmc include the right libraries. Pick the "language" according to this table:

- **Charm++** if your program includes Charm++, Charm, C++, and C.
- **Converse** if your program includes C or C++.
- **sdag** if your program includes structured dagger.
- **idl** if your program includes IDL bindings for Charm++.

`-tracemode` ***tracing-mode***: Selects the desired degree of tracing for Charm and Charm++ programs. See the Charm manual and the Projections and SummaryTool manuals for more information. Currently supported modes are `none`, `summary`, and `projections`. Default is `-tracemode none`.

A.2 Executing Converse/Charm/Charm++ Programs

The Charm linker produces one executable file. On machines with a host (such as a network of workstations), a link to the proper host program **conv-host** is created in the user program directory. Sample execution examples are given below (the executable is called pgm). Exact details will differ from site to site. The list of Charm command line options is in Section A.2.1.

- **ASCI Red:**

  yod -sz 4 pgm

  runs pgm on four processors.

- **Cray T3E:**

  mpprun -n 4 pgm

  runs pgm on four processors.

- **SGI Origin2000 (origin-mpi):**

  mpirun -np 4 pgm

  runs pgm on four processors.

- **SGI Origin2000 (origin2000 or origin-pthreads):**

  pgm +p4

  runs pgm on four processors.

- **Network of workstations:**

  conv-host pgm +p4

executes pgm on 4 nodes. In a network environment, Charm must be able to locate the directory of the executable. If all workstations share a common file name space this is trivial. If they don't, Charm will attempt to find the executable in a directory with the same path from the **$HOME** directory. Pathname resolution is performed as follows:

1. The system computes the absolute path of **pgm**.
2. If the absolute path starts with the equivalent of **$HOME** or the current working directory, the beginning part of the path is replaced with the environment variable **$HOME** or the current working directory. However, if exec_home is specified in the nodes file (see below), the beginning part of the path is replaced with exec_home.
3. The system tries to locate this program (with modified pathname and appended extension if specified) on all nodes.

The list of nodes must be specified in a file. The format of this file allows you to define groups of machines, giving each group a name. Each line of the nodes file is a command. The most important command is:

```
host <hostname> <qualifiers>
```

which specifies a host. The other commands are qualifiers: they modify the properties of all hosts that follow them. The qualifiers are:

`group <groupname>`	- subsequent hosts are members of specified group
`login <login>`	- subsequent hosts use the specified login
`shell <shell>`	- subsequent hosts use the specified remote shell
`setup <cmd>`	- subsequent hosts should execute cmd
`home <dir>`	- subsequent hosts should find programs under dir
`cpus <n>`	- subsequent hosts should use N light-weight processes
`speed <s>`	- subsequent hosts have relative speed rating
`ext <extn>`	- subsequent hosts should append extn to the pgm name

Note: By default, conv-host uses a remote shell "rsh" to spawn node processes on the remote hosts. The `shell` qualifier can be used to override it with say, "ssh". One can set the `CONV_RSH` environment variable or use conv-host option `++remote-shell` to override the default remote shell for all hosts with unspecified `shell` qualifier.

All qualifiers accept "*" as an argument, this resets the modifier to its default value. Note that currently, the passwd, cpus, and speed factors are ignored. Inline qualifiers are also allowed:

```
host beauty ++cpus 2 ++shell ssh
```

Except for "group", every other qualifier can be inlined, with the restriction that if the "setup" qualifier is inlined, it should be the last qualifier on the "host" or "group" statement line.

Here is a simple nodes file:

```
group kale-sun ++cpus 1
  host charm.cs.uiuc.edu ++shell ssh
  host dp.cs.uiuc.edu
```

```
  host grace.cs.uiuc.edu
  host dagger.cs.uiuc.edu
group kale-sol
  host beauty.cs.uiuc.edu ++cpus 2
group main
  host localhost
```

This defines three groups of machines: group kale-sun, group kale-sol, and group main. The ++nodegroup option is used to specify which group of machines to use. Note that there is wraparound: if you specify more nodes than there are hosts in the group, it will reuse hosts. Thus,

```
conv-host pgm ++nodegroup kale-sun +p6
```

uses hosts (charm, dp, grace, dagger, charm, dp) respectively as nodes (0, 1, 2, 3, 4, 5).

If you don't specify a ++nodegroup, the default is ++nodegroup main. Thus, if one specifies

```
conv-host pgm +p4
```

it will use "localhost" four times. "localhost" is a Unix trick; it always find a name for whatever machine you're on.

Since the new nodes file is incompatible with the old nodes file, it has been renamed. It now is called ".nodelist", and all the options and environment variables pertaining to it have also been renamed `NODELIST`.

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

Note that the Charm linker will provide the correct executable. The user, however, needs to know how programs are run for the particular machine.

A.2.1 Command Line Options

A Charm program accepts the following command line options:

+pN Run the program with N processors. The default is 1. Note that on some nonshared memory machines, e.g., nCUBE/2, the user must specify the number of processors using the command provided for that machine (e.g. **xnc** on the nCUBE/2). In such cases the **+p** option is ignored.

+ss Print summary statistics about chare creation. This option prints the total number of chare creation requests, and the total number of chare creation requests processed across all processors.

+cs Print statistics about the number of create chare messages requested and processed, the number of messages for chares requested and processed, and the number of messages for branch office chares requested and processed, on a per processor basis. Note that the number of messages created and processed for a particular type of message on a given node may not be the same, since a message may be processed by a different processor from the one originating the request.

user_options Options that are be interpreted by the user program may be included after all the system options. However, **user_options** cannot start with +. The **user_options** will be passed as arguments to the user program via the usual **argc/argv** construct to the **main** entry point of the main chare. Charm system options will not appear in **argc/argv**.

A.2.2 Additional Uniprocessor Command Line Options

The uniprocessor versions can be used to simulate multiple processors on a single workstation. Any number of processors between 1 and 32 can be simulated by using the **+p** option, limited only by the available memory on the uniprocessor workstation. By default, the uniprocessor versions handle a single message from each processor, going in order from processor 0 thru $P-1$ (where P is the number of processors) repeatedly.

A.2.3 Additional Network Command Line Options

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

++debug Run each node under gdb in an xterm window, prompting the user to begin execution.

++debug-no-pause Run each node under gdb in an xterm window immediately (i.e. without prompting the user to begin execution).

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

++resend-wait Timeout before retransmitting datagrams (in msec).

++resend-fail Timeout before retransmission fails (in msec). This parameter can help the user kill "runaway" processes, which may not be killed otherwise when the user interrupts the program before it completes execution. Currently a bug exists in the network version that may cause programs to terminate prematurely if this value is set too low and **scanf** operations are being performed.

++nodelist File containing list of nodes.

If using the **++debug** option, the user must ensure the following:

1. **xterm**, **xdpyinfo**, and **gdb** must be in the user's path.
2. The path must be set in the **.cshrc** file, not the **.login** file, because **rsh** does not run the **.login** file.
3. The nodes must be authorized to create windows on the host machine (see man pages for **xhost** and **xauth**).

B Charm++ Keywords

The following is the complete list of keywords in Charm++:

1. array
2. char
3. chare
4. class
5. double
6. entry
7. exclusive
8. extern
9. float
10. group
11. int
12. long
13. mainchare
14. mainmodule
15. message
16. module
17. nodegroup
18. packed
19. readonly
20. short
21. stacksize
22. sync
23. template
24. thisgroup
25. thishandle
26. threaded
27. unsigned
28. varsize

29. virtual
30. void

The following is the complete list of system operators and calls in CHARM++. Currently no user-defined functions may have one of these names.

1. CkAllocBuffer
2. CkAllocMsg
3. CkAllocSysMsg
4. CkArgMsg
5. CkArrayID
6. CkBroadcastMsgBranch
7. CkBroadcastMsgNodeBranch
8. CkChareID
9. CkCopyMsg
10. CkCreateChare
11. CkCreateGroup
12. CkCreateNodeGroup
13. CkError
14. CkExit
15. CkFreeMsg
16. CkFreeSysMsg
17. CkGetChareID
18. CkGetGroupID
19. CkGetNodeGroupID(void);
20. CkGetRefNum
21. CkGetSrcNode
22. CkGetSrcPe
23. CkLocalBranch(int groupID);
24. CkLocalNodeBranch(int groupID);
25. CkMyPe
26. CkMyRank

27. CkMyNode
28. CkNodeFirst
29. CkNodeOf
30. CkNodeSize
31. CkNumNodes
32. CkNumPes
33. CkPrintf
34. CkPriorityPtr
35. CkQdMsg
36. CkRankOf
37. CkRegisterChare
38. CkRegisterEp
39. CkRegisterMainChare
40. CkRegisterMsg
41. CkRegisterReadonly
42. CkRegisterReadonlyMsg
43. CkRemoteCall
44. CkRemoteBranchCall
45. CkRemoteNodeBranchCall
46. CkScanf
47. CkSendMsg
48. CkSendMsgBranch
49. CkSendMsgNodeBranch
50. CkSendToFuture
51. CkSetQueueing
52. CkSetRefNum
53. CkStartQD
54. CkTimer
55. CkWaitQd
56. CkWallTimer

C Syntax Changes from CHARM++ 4.9

The following changes are required to make older CHARM++ 4.9 programs run with the new translator and runtime system in CHARM++ 5.0.

- Replace all references to `*.top.h` and `*.bot.h` to `*.decl.h` and `*.def.h` respectively. This should be done in Makefile, and everywhere these two types of files are included.
- Change all X.ci files to include a top-level enclosure of module X ....
- Change **chare** main to **mainchare** main.
- Replace **chare_object** by **Chare**.
- Replace **groupmember** by **Group**.
- Replace **comm_object** by CMessage_¡msgName¿ in all message declarations in `*.h` files.
- Remove all **operator new** methods of messages.
- Remove **MsgIndex(..)** parameter to **new** for message allocation.
- Remove all the empty messages (used only for triggering computations.) from `*.h`, `*.ci`, `*.C` files. All the entry methods that take these empty messages as parameters should be made methods with **void** parameter. This should be done in all `*.ci`, `*.h`, and `*.C` files. In these methods, there may be a `delete msg`. Remove that.
- Check for **mainhandle** in the source. If it is there, declare it as a **readonly** variable in `.ci` file, and initialize it in the **mainchare**'s constructor, so that it is available to all the processors during the run.
- All the **packmessage** declarations in `*.ci` files should be changed to **message** [**packed**].
- All **CPrintf**, **CScanf**, and **CError** should be changed to **CkPrintf**, **CkScanf**, and **CkError**.
- All **CharmExit** should be changed to **CkExit**.
- Replace **CMyPe** by **CkMyPe**. Replace **CNumPes** by **CkNumPes**.
- Change **ChareIDType** to **CkChareID**.
- Change signature of M::pack and M::unpack for all messages M in `*.h`, `*.C` to
  `static void *pack(M* msg)`
  and
  `static M* unpack(void *buf)`
  and change the code accordingly. This is a significant change because `pack` and `unpack` used to be instance methods, and now they are class static methods. Avenues of optimizations open with this change, but one need not explore those in the interest of time immediately. Further, one should make sure that the performance of the new scheme is at least as good as the old one.
- Replace all **new_group** by CProxy_¡grpName¿::ckNew.
- Replace **new_chare2** by proxy creation.

- Replace **CSendMsg** by temporary proxy creation based on ChareID and invoking appropriate method on it. One optimization is to create a proxy immediately after a ChareID is received, and reusing it everytime.
- Similarly replace **CSendMsgBranch** by temporary proxy creation based on the groupID, and invoking appropriate method on it, with processor number as the second parameter. Once again, there is an opportunity for optimization here.
- Similarly replace **CBroadcastMsgBranch** by temporary proxy creation based on the groupID, and invoking appropriate method on it, without any second parameter. Once again, there is an opportunity for optimization here.
- Replace **CStartQuiescence** by **CkStartQD**.
- Replace **GetEntryPtr** by appropriate static method (**ckIdx_***) calls.
- Replace **CLocalBranch** macros by **ckLocalBranch** instance method on temporarily created proxy.
- Change **CPriorityPtr** to **CkPriorityPtr**, also cast it explicitly to `(int *)`.
- Replace **QuiescenceMessage** by **CkQDMsg**. Remove all extern declarations of **CkQDMsg** from `*.ci` files.

D 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:

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 cutoff 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 1 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

```
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 1: Compute Object in a Molecular Dynamics Application

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 1 with a thread-based implementation of the same scheme in figure 2. 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, 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.

Structured Dagger [3] 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

```
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 2: Compute Thread in a Molecular Dynamics Application

```
class compute_object
  sdagentry 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 3: Structured Dagger Implementation of the Compute Object

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.

D.1 Grammar

D.1.1 Tokens

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

D.1.2 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> ")"
```

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

E Interface Description Language (IDL)

The Interface Description Language(IDL) is the language used to describe the interfaces that client objects call and server object implementations provide.

A typical simple IDL interface file follows:

```
module myModule {
  interface myClass1 {
    attribute long myAttr1;
    attribute int myAttr2;
    oneway void myMethod1(
        in char c,
        in short s,
        in long l,
        in unsigned short us,
        in unsigned long ul,
        out float f,
        inout double d,
  out int a[10]);
  };
};
```

An interface can have attributes as well as operations. Array parameters in operation declaration are supported. Templates and String types are currently not supported.

Parameter declaration:

Parameter declaration attributes include:

- **in** - the parameter is passed from client to server

- **out** - the parameter is passed from server to client
- **inout** - the parameter is passed in both directions

How to write client/server side code :

To define the classes declared in the `.idl` file, one must write `.h` and `.C` file for the classes as usual. For example, in the `.idl` file, we have:

```
module myModule {
  interface myClass1 {
    oneway void myMethod1( in char c);
  };
};
```

In the `.h` file:

```
class myClass1 {
  public:
    void myMethod1( char );
    myClass1();
    ~myClass1();
};
```

We implement these methods in the `.C` file.

Instead of using `myClass1` directly, one should declare and use the class by using the name `CImyClass1`. For example, in `main.C`:

```
main::main(CkArgMsg* m)
{
  CImyClass1 a;
  // create the object
  a.ciSetProc(CI_PE_ANY).ciCreate();
  // call method
  a.myMethod1('a');
  // delete the object
  a.ciDelete();
}
```

F Further Information

F.1 Related Publications

For starters, see the publications and reports as well as related manuals that can be found on the Parallel Programming Laboratory website: `http://charm.cs.uiuc.edu/`.

Some CHARM++ related papers: [6],[9]. Papers on CONVERSE: [4], [5]. Some CHARM related papers that formed the early foundation for CHARM++: [7], [8], [2]. Papers on Structured Dagger and Dagger: [1], [3]. Projections papers: [11], [10].

F.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.

F.2.1 Projections

Projections [11], [10] 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.

F.2.2 TeMPO

TeMPO (Threaded Message Passing Objects) provides a simple way to communicate between threaded methods of objects. Using TeMPO, chares, groups, and arrays can have threads associated with them. Any CHARM++ object can send a tagged message to a TeMPO Object, and a threaded method in a TeMPO Object can block on a specific tagged message.

In order to use TeMPO functionalities, chares, groups, and arrays need to inherit from TeMPO objects called **TempoChare**, **TempoGroup**[15], and **TempoArray** respectively. This inheritance also needs to be specified in the CHARM++ interface file. Thus, a chare C in module M that needs to receive a tagged message directed to it in entry E, needs to inherit from **TempoChare** as shown below:

```
// file.ci
module M {
 ...
 chare C : TempoChare {
   entry C(void);
   entry [threaded] void E(Msg *); // note use of threaded
 };
 ...
}

// file.h
#include "file.decl.h"
...
class C : public TempoChare { // TempoChare inherits from Chare already
 public:
```

[15]Currently, a nodegroup variety of TeMPO object does not exist, thoug it will be added in future.

```
  C(void);
  E(Msg *);
};
...
```

The code for entry method `E` can contain a call to **ckTempoRecv** to block for a tagged message. Any ordinary chare can send a tagged message to a TeMPO object using **ckTempoSend** static method of **TempoChare**.

TempoGroup and **TempoArray** provide additional static methods to send tagged messages to individual elements, as well as broadcast. In addition, **TempoArray** provides methods to perform "reduction" over all the elements of an array. Currently supported reduction operations are max, min, sum, and product over datatypes float, int, and double. The signatures of these methods are given below:

```
class TempoChare : public Chare, public ... {
  void ckTempoRecv(int tag, void *buffer, int buflen);
  void ckTempoRecv(int tag1, int tag2, void *buffer, int buflen);
  static void ckTempoSend(int tag1, int tag2, void *buffer,int buflen,
                          CkChareID cid);
  static void ckTempoSend(int tag, void *buffer,int buflen, CkChareID cid);
  int ckTempoProbe(int tag1, int tag2);
  int ckTempoProbe(int tag);
};

// TempoGroup includes all the TempoChare methods

class TempoGroup : public Group, public ... {
  static void ckTempoBcast(int tag, void *buffer, int buflen, CkGroupID grpid);
  static void ckTempoSendBranch(int tag1, int tag2, void *buffer, int buflen,
                                CkGroupID grpid, int processor);
  static void ckTempoSendBranch(int tag, void *buffer, int buflen,
                                CkGroupID grpid, int processor);
  void ckTempoBcast(int sender, int tag, void *buffer, int buflen);
  void ckTempoSendBranch(int tag1, int tag2, void *buffer, int buflen,
                         int processor);
  void ckTempoSendBranch(int tag, void *buffer, int buflen, int processor);
};

// TempoArray includes all the TempoChare methods

class TempoArray : public ArrayElement, public Tempo
{
  static void ckTempoSendElem(int tag1, int tag2, void *buffer, int buflen,
                              CkArrayID aid, int idx);
  static void ckTempoSendElem(int tag, void *buffer, int buflen,
                              CkArrayID aid, int idx);
  void ckTempoSendElem(int tag1, int tag2, void *buffer, int buflen, int idx);
  void ckTempoSendElem(int tag, void *buffer, int buflen, int idx);
  void ckTempoBarrier(void);
```

```
  void ckTempoBcast(int sender, int tag, void *buffer, int buflen);
  void ckTempoReduce(int root, int op, void *inbuf, void *outbuf, int count,
                     int type);
  void ckTempoAllReduce(int op,void *inbuf,void *outbuf,int count,int type);
};
```

All the `ckSend*` methods have versions that send messages with one tag or with two tags. A wild card tag (**TEMPO_ANY** may be in specified in **ckTempoRecv** that matches with any tag. All the tags must range between 0 and 1024. All the other tags have other uses in the system. Reduction operations are indicated by integer constants. They are: **TEMPO_MAX**, **TEMPO_MIN**, **TEMPO_SUM**, and **TEMPO_PROD**. Reduction data types can be specified using integer constants: **TEMPO_FLOAT**, **TEMPO_INT**, and **TEMPO_DOUBLE**. The root parameter in **ckTempoReduce**, and the sender parameter in **ckTempoBcast** indicate whether the calling element is the root of the collective operation or not. In case of a reduction, the root element is returned the result of the reduction operation in outbuf[16] In case of a broadcast, buffer on the sender contains the message to be sent to other elements. For each of the send methods, the specified message buffer could be reused once the method returns.

All the TeMPO include files are automatically includes from `charm++.h`. There is no need to include any other file to use TeMPO.

F.2.3 Communication

Communication optimizations tend to be specific to a particular architecture or an application. To improve portability and to reduce the cost of developing parallel applications a mechanism to integrate these different optimizations should exist. Moreover, it should be possible to automatically adapt the strategy to the situation at hand. The communication library integrates the different strategies to perform each-to-many multicast, including tree-based multicast, grid -based multicast and hypercube-based (dimensional exchange) schemes. The framework provided is flexible enough to absorb new strategies and communication patterns. It also provides the capability to do dynamic switching of strategies. This helps the library to adapt itself to the existing environment.

F.2.4 Barrier

We needed a way to synchronise a subset of tasks in CHARM++ programs. Quiescence detection is one way this can be done, if we want to synchronize **all** tasks, but barriers provide a more flexible option with less overhead.

To make use of barriers, the user initializes a *barrier group* in the main function. A virtual binary tree structure is imposed on the PEs, with PE 0 at the root. PEs perform computations and then execute `atBarrier` when they want to synchronize. `atBarrier` takes a single parameter of type `FP *`. The user must declare and initialize some variable *fnptr* as follows:

```
  FP *fnptr = new FP;

  fnptr->fp = theFn;
```

The variable *theFn* above is the void function to be executed when all PEs have synchronized.

[16] outbuf and inbuf can be the same.

The barrier keeps track of child PEs, and when a given PE has heard from all its children, it then notifies the parent. When the root hears from both children, all PEs are accounted for, and the respective void function is executed on each PE.

What follows is a very simple test program illustrating the usage of a barrier.

`test.ci`:

```
mainmodule Test {
  extern module Barrier;
  readonly CkChareID mainhandle;
  readonly int barrierGroup;

  mainchare main {
    entry main();
    entry void Quiescence1(void);
  };

  group busy {
    entry busy(void);
  };
};
```

`test.h`:

```
#include "Test.decl.h"
int barrierGroup;
CkChareID mainhandle;

class main : public Chare {
public:
  main(CkArgMsg *m);
  void Quiescence1(void);
};

class busy : public Group {
public:
  busy(void);
}
```

`test.C`:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "charm++.h"

#include "test.h"
#include "Test.def.h"
```

```
#include "barrier.h"

main::main(CkArgMsg *m)
{
  barrierGroup = barrierInit();
  CProxy_busy::ckNew();
  CkStartQD(CProxy_main::ckIdx_Quiescence1(), &mainhandle);
}

void main::Quiescence1()
{
  CkPrintf("All done... Exiting.\n");
  CkExit();
}

void theFn();

busy::busy()
{
  int i, j=0;
  FP *fnptr = new FP;

  for (i=0; i<(CkMyPe()+1)*1000000; i++)
    j = j + i;
  CkPrintf("[%d] going to barrier with j=%d\n", CkMyPe(), j);
  fnptr->fp = theFn;
  CProxy_barrier(barrierGroup).ckLocalBranch()->atBarrier(fnptr);
}

void theFn()
{
  CkPrintf("[%d] fnptr executing!\n", CkMyPe());
}
```

F.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 CHARM/CHARM++ and information on how to contact the Parallel Programming laboratory, see our website at *http://charm.cs.uiuc.edu/*. The software is also available by anonymous ftp, from a.cs.uiuc.edu, under the directory pub/research-groups/CHARM.

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

A mailing list exists for announcements about software releases and updates relating to CHARM++/CONVERSE. To subscribe, send e-mail to: ???????.

References

[1] A. Gursoy and L.V. Kalé. Dagger: Combining the Benefits of Synchronous and Asynchronous Communication Styles. In H. G. Siegel, editor, *Proceedings of the 8th International Parallel Processing Symposium*, pages 590–596, Cancun, Mexico, April 1994.

[2] L. V. Kale. Parallel programming with charm: An overview. Technical report, Parallel Programming Laboratory, University of Illinois at Urbana-Champaign, July 1993.

[3] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[4] L. V. Kale, Milind Bhandarkar, Robert Brunner, and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[5] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[6] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[7] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[8] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[9] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[10] L.V. Kalé and Amitabh Sinha. Projections : A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Sympos ium*, pages 108–114, April 1993.

[11] L.V. Kalé and Amitabh Sinha. Projections: A preliminary performance tool for charm. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, Newport Beach, CA, April 1993.

Index

++debug, 49
++debug-no-pause, 49
++maxrsh, 49
++nodelist, 49
++resend-fail, 49
++resend-wait, 49
.nodes, 49

abstract base class, 43
alloc, 16
arbitrary-precision priorities, 20
array, 7, 10, 28, 29
arrays, 28

base chare, 42
bitvector format, 20
bitvector priorities, 19
branch, 26–28
broadcast, 26

chare, 4, 6, 7, 10, 12, 14, 21–26, 38–40
chare array, 28
chare arrays, 10
chare group, 23
chare groups, 10
CK_PE_ANY, 23
CK_QUEUEING_BFIFO, 19
CK_QUEUEING_BLIFO, 19
CK_QUEUEING_FIFO, 19
CK_QUEUEING_IFIFO, 19
CK_QUEUEING_ILIFO, 19
CK_QUEUEING_LIFO, 19
CkAbort, 41
CkAllocBuffer, 14
CkAllocMsg, 17
CkChareID, 22, 24
ckerr, 40
CkError, 40
CkExit, 7, 41
CkFreeMsg, 15
CkGroupID, 25–27
ckin, 40
CkLocalBranch, 26
ckLocalBranch, 27
CkLocalNodeBranch, 28
CkMyNode, 40
CkMyNodes, 41
CkMyPe, 40
CkMyRank, 40
CkNodeFirst, 41
CkNodeOf, 41
CkNodeSize, 41
CkNumPes, 40
ckout, 40
CkPrintf, 40
CkPriorityPtr, 19
CkRankOf, 41
CkScanf, 40
CkSetQueueing, 19, 20
CkStartQd, 39
CkTimer, 41
CkWaitQd, 39
CkWallTimer, 41
command line options, 48
constructor, 23
CProxy, 39
CProxy_, 23

default constructor, 43
delete, 14
derived chare, 42

entry method, 4, 7, 21–24, 39
entry methods, 22
EntryIndex, 39
exclusive, 21, 27, 28

group, 7, 10, 12, 24–26

handle, 22, 23

inheritance, 42
input/output, 39, 40
integer priorities, 19
integer priority range, 20

load balancing, 23

main, 49
message, 4, 7, 10, 12–14, 22–24, 40
message delivery order, 18, 20
message inheritance, 43
message packing, 4, 14
message priority, 19
message serialization, 14
migrate, 32
module, 10, 44
multiple inheritance, 43

multiple priority types, 20

network command line options, 49
new, 13, 15
node groups, 27
nodegroup, 7, 12, 27, 28
nodes file, 49

pack, 14, 16
packed, 14
packed messages, 14, 16
priorities, 18, 19
prioritized execution, 18
prioritized message passing, 18
priority, 14
priority field, 19
priority pointer, 19
priority range splitting, 20
proxy, 24, 26
pure virtual, 43

queueing types, 19
quiescence, 4, 39

readonly, 10, 38
reduction, 34

seed, 23
sequential objects, 21
serialized messages, 14
sync, 21

template, 44
templates, 43
thisgroup, 25, 27
thishandle, 22, 25, 27
threaded, 7, 21
timers, 41

uniprocessor command line options, 49
unpack, 14, 16

variable size messages, 16
varsize, 16
varsize message, 13, 16
virtual, 21, 43
virtual handle, 23, 25