Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

# The Charm++ Programming Language Manual

Version 6.4.0

# Contents

# Chapter 1

# Programming and Execution Model

Charm++ is a C++-based parallel programming system, founded on the migratable-objects programming model, and supported by a novel and powerful adaptive runtime system. It supports both irregular as well as regular applications, and can be used to specify task-parallelism as well as data parallelism in a single application. It automates dynamic load balancing for task-parallel as well as data-parallel applications, via separate suites of load-balancing strategies. Why are its message-driven execution model, it supports automatic latency tolerance, among other features. Charm++ also supports automatic checkpoint/restart checkpoints, as well as fault tolerance based on distributed checkpoints.

Charm++ is a production-quality parallel programming system used by multiple applications in science and engineering on supercomputers as well as departmental clusters around the world. Currently the parallel platforms supported by Charm++ are the BlueGene/L,BlueGene/P, BlueGene/Q, Cray XT, XE and XK series (including XK6 and XE6), a single workstation or a network of workstations (including x86 (running Linux, Windows, MacOS)), etc. The communication protocols and infrastructures supported by Charm++ are UDP, TCP, Myrinet, Infiniband, uGNI, and PAMI. 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.

**Migratable Objects Programming Model**

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

**Charm++ Execution Model**

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

Methods of a chare that can be remotely invoked are called *entry* methods. Entry methods may take marshalled parameters, or a pointer to a message object. Since chares can be created on remote processors, obviously some constructor of a chare needs to be an entry method. Ordinary entry methods[1] are completely

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

non-preemptive– Charm++ will never interrupt an executing method to start any other work, and all calls made are asynchronous.

Charm++ provides dynamic seed-based load balancing. Thus location (processor number) need not be specified while creating a remote chare. The Charm RTS 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 RTS 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 behavior is used by the dynamic load-balancing framework for chare containers, such as arrays.)

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

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

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

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

Charm++ program execution is terminated by the CkExit call. Like the exit system call, CkExit never returns. The Charm RTS ensures that no more messages are processed and no entry methods are called after a CkExit. CkExit need not be called on all processors; it is enough to call it from just one processor at the end of the computation.

The following sections provide detailed information about various features of the Charm++ programming system.[2].

### 1.0.1 Basic Constructs and Program Structure

Charm++ is an object-based parallel programming paradigm. 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:

[2]For a description of the underlying design philosophy please refer to the following papers :
L. V. Kale and Sanjeev Krishnan, *"Charm++: Parallel Programming with Message-Driven Objects"*, in "Parallel Programming Using C++", MIT Press, 1995.
L. V. Kale and Sanjeev Krishnan, *"Charm++: A Portable Concurrent Object Oriented System Based On C++"*, Proceedings of the Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), September 1993.

- Concurrent Objects : called *chares*[3]
- Communication Objects : Messages
- Readonly data

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

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

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.

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

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

CHARM++ is built on top of C++, and also based on "encapsulation". Similar to C++, CHARM++ entities can contain private data, and public methods. The major difference is that these methods can be invoked from remote processors asynchronously. Asynchronous method invocation means that the caller does not wait for the method to be actually executed and does not wait for the method's return value. Therefore, CHARM++ methods (called entry methods) do not have a return value[4]. Since the actual CHARM++ object on which the method is being invoked may be on a remote processor[5], the C++ way of referring to an object, via a pointer, is not valid in CHARM++. Instead, we refer to a remote chare via a "proxy", as explained below.

### 1.0.2 Proxies and Handles

Those familiar with various component models (such as CORBA) in the distributed computing world will recognize "proxy" to be a dummy, standin entity that refers to an actual entity. For each chare type, a "proxy" class exists.[6] The methods of this "proxy" class correspond to the remote methods of the actual class, and act as "forwarders". That is, when one invokes a method on a proxy to a remote object, the proxy forwards this method invocation to the actual remote object. All entities that are created and manipulated

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

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

[5] With its own, different address space

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

remotely in Charm++ have such proxies. Proxies for each type of entity in Charm++ have some differences among the features they support, but the basic syntax and semantics remain the same – that of invoking methods on the remote object by invoking methods on proxies.

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

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

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

### 1.0.3 Entities in Charm++ programs

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

#### Sequential Objects

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

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

#### Messages

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

#### Chares

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

### Chare Arrays

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

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

### Chare Groups

Chare Groups[7] are a special type of concurrent objects. Each chare group is a collection of chares, with one representative (group member) on each processor. All the members of a chare group share a globally unique name (handle, defined by Charm RTS 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 RTS has to be notified that these chares are semantically different, and therefore chare groups have a different declaration in the interface specification file.

### Chare Nodegroups

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

### Entry Methods

In CHARM++, chares, groups and nodegroups communicate using remote method invocation. These "remote entry" methods may either take marshalled parameters, described in the next section; or special objects called messages.

CHARM++ programs are open to the full gamut of program design techniques that are possible in C++. One may view the role of the programming model as providing a platform for addressing and interacting with remote objects. However, CHARM++does not provide a full global address space. Each process in the parallel execution has its own address space and CHARM++does not implicitly share or synchronize global or static variables.

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

# Chapter 2

# Machine Model

node, PE, ranks?, smp-mode and non-smp mode terminology. etc.

# Part I

# Basic Usage

# Chapter 3

# Program Structure, Compilation and Utilities

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

## 3.1 .ci Files

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

## 3.2 Modules

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

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

## 3.3 Generated Files

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

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

$\boxed{\beta}$

It should be noted that the generated files have no dependence on the name of the *ci* file, but only on the names of the modules. This can make automated dependency-based build systems slightly more complicated. We adopt some conventions to ease this process. This is described in **??**.

## 3.4 Module Dependencies

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

```
module mySecondModule {

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

    // More parallel interface declarations
    ...
};
```

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

## 3.5 The Main Module and Reachable Modules

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

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

```
module A {
    ...
};

module B {
    extern module A;
    ...
};

module C {
    extern module A;
    ...
};

module D {
    extern module B;
    ...
};

module E {
    ...
};

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

## 3.6 Including other headers

There can be occasions where code generated from the module definitions requires other declarations / definitions in the user program's sequential code. Usually, this can be achieved by placing such user code before the point of inclusion of the decl.h file. However, this can become laborious if the decl.h file has to included in several places. Charm++ supports the keyword include in *ci* files to permit the inclusion of any header directly into the generated decl.h files.

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

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

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

## 3.7 The main() function

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

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

## 3.8 Compiling Charm++ Programs

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

### 3.8.1 Utility Functions

The following calls provide basic rank information and utilities useful when running a Charm++ program. Other utilities are listed in Section 16.0.4.

int CkNumPes()

returns the total number of processors, across all nodes.

int CkMyPe()

returns the processor number on which the call was made.

void CkAssert(int expression)

Aborts the program if expression is 0

void CkAbort(const char *message)

Cause the program to abort, printing the given error message. This routine never returns.

void CkExit()

This call informs the Charm RTS that computation on all processors should terminate. After the currently executing entry method completes, no more messages or entry methods will be called on any other processor. This routine never returns.

double CkWallTimer()

Returns the elapsed time in seconds since the program has started from the wall clock timer.

#### Terminal I/O

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

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

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

int CkPrintf(format [, arg]*)

This call is used for atomic terminal output. Its usage is similar to `printf` in C. However, CkPrintf has some special properties that make it more suited for parallel programming on networks of workstations. CkPrintf routes all terminal output to the charmrun, which is running on the host computer. So, if a chare on processor 3 makes a call to CkPrintf, that call puts the output in a TCP message and sends it to host computer where it will be displayed. This message passing is an asynchronous send, meaning that the call to CkPrintf returns immediately after the message has been sent, and most likely before the message has actually been received, processed, and displayed. [1]

void CkError(format [, arg]*))

Like CkPrintf, but used to print error messages on `stderr`.

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

int CkScanf(format [, arg]*)

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

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

# Chapter 4

# Basic Syntax

### 4.0.1 Entry Methods

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Marshalling User-Defined Structures and Classes**

The marshalling system uses the pup framework to copy data, meaning every user class that is marshalled needs either a pup routine, a "PUPbytes" declaration, or a working operator—. See the PUP description in Section 7 for more details on these routines.

Any user-defined types in the argument list must be declared before including the ".decl.h" file. As usual in C++, it is often dramatically more efficient to pass a large structure by reference than by value.

### 4.0.2 Chare Objects

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

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

Although it is *declared* in an interface file, a chare is a C++ object and must have a normal C++ *implementation* (definition) in addition. A chare class `ChareType` must inherit from the class `CBase_ChareType`, which is a special class that is generated by the Charm++translator from the interface file.

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

```
class ChareType : public CBase_ChareType {
    // Data and member functions as in C++
    // One or more entry methods definitions of the form:
    public:
        ChareType(parameters2);
        void EntryMethodName2(parameters2);
};
```

Each chare encapsulates data associated with medium-grained units of work in a parallel application. Chares can be dynamically created on any processor; there may be thousands of chares on a processor. The location of a chare is usually determined by the dynamic load balancing strategy. However, once a chare commences execution on a processor, it does not migrate to other processors[1]. Chares do not have a default "thread of control": the entry methods in a chare execute in a message driven fashion upon the arrival of a message[2].

The entry method definition specifies a function that is executed *without interruption* when a message is received and scheduled for processing. Only one message per chare is processed at a time. Entry methods are defined exactly as normal C++ function members, except that they must have the return value `void` (except for the constructor entry method which may not have a return value, and for a *synchronous* entry method, which is invoked by a *threaded* method in a remote chare). Each entry method can either take no arguments, take a list of arguments that the runtime system can automatically pack into a message and send (see section **??**), or take a single argument that is a pointer to a Charm++message (see section 10.0.2).

A chare's entry methods can be invoked via *proxies* (see section 1.0.2). Proxies to a chare of type `chareType` have type `CProxy_chareType`. By inheriting from the CBase parent class, each chare gets a `thisProxy` member variable, which holds a proxy to itself. This proxy can be sent to other chares, allowing them to invoke entry methods on this chare.

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

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

### Chare Creation

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

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

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

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

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

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

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

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

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

### Method Invocation on Chares

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

chareProxy.EntryMethod(*parameters*)

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

### Local Access

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

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

## 4.0.3 Read-only Variables, Messages and Arrays

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

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

```
readonly Type ReadonlyVarName;
```

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

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

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

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

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

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

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

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

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

# Chapter 5

# Chare Arrays

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

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

### 5.0.1 Declaring a One-dimensional Array

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

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

Array elements extend the system class CBase_*ClassName*, inheriting several fields:

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

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

    void someEntry(parameters2);
};
```

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

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

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

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

See the section 7.0.3 on migratable array elements for more information on the migration constructor that takes CkMigrateMessage * as the argument.

### 5.0.2 Declaring Multi-dimensional Arrays

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

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

The last declaration expects an array index of type CkArrayIndex*Foo*, which must be defined before including the `.decl.h` file (see section 11 on user-defined array indices for more information).

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

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

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

### 5.0.3 Creating an Array

An array is created using the CProxy_Array::ckNew routine. This returns a proxy object, which can be kept, copied, or sent in messages. The following creates a 1D array containing elements indexed (0, 1, . . . , *dimX*-1):

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

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

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

For 4D, 5D, 6D and user-defined arrays, this functionality cannot be used. The array elements must be inserted individually as described in section 11.

During creation, the constructor is invoked on each array element. For more options when creating the array, see section 11.

### 5.0.4 Entry Method Invocation

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

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

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

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

You may invoke methods on array elements that have not yet been created. The Charm++ runtime system will buffer the message until the element is created. [1]

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

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

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

### 5.0.5 Broadcasts on Chare Arrays

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

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

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

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

### 5.0.6 Reductions on Chare Arrays

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

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

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

This call contributes nBytes bytes starting at data to the reduction type (see Section 5.0.7). Unlike sending a message, you may use data after the call to contribute. All members of the chare array or group must call contribute, and all of them must use the same reduction type.

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

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

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

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

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

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

Typically the client entry method of a reduction takes a single argument of type CkReductionMsg (see Section 13.1.4). However, by giving an entry method the reductiontarget attribute in the `.ci` file, you can instead use entry methods that take arguments of the same type as specified by the *contribute* call. When creating a callback to the reduction target, the entry method index is generated by `CkReductionTarget(ChareClass, method_name)` instead of `CkIndex_ChareClass::method_name(...)`. For example, the code for a typed reduction that yields an `int`, would look like this:

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

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

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

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

This will also work for arrays of data elements(`entry [reductiontarget] void done(int n, int result[n])`), and for any user-defined type with a PUP method (see 7). If you know that the reduction will yield a particular number of elements, say 3 `int`s, you can also specify a reduction target which takes 3 `int`s and it will be invoked correctly.

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

```
contribute();
```

In all cases, the result of the reduction operation is passed to the *reduction client*. Many different kinds of reduction clients can be used, as explained in Section 13.1.4.

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

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

### 5.0.7 Built-in Reduction Types

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

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

1. CkReduction::nop– no operation performed.
2. CkReduction::sum_int, sum_float, sum_double– the result will be the sum of the given numbers.
3. CkReduction::product_int, product_float, product_double– the result will be the product of the given numbers.
4. CkReduction::max_int, max_float, max_double– the result will be the largest of the given numbers.
5. CkReduction::min_int, min_float, min_double– the result will be the smallest of the given numbers.
6. CkReduction::logical_and– the result will be the logical AND of the given integers. 0 is false, nonzero is true.
7. CkReduction::logical_or– the result will be the logical OR of the given integers.
8. CkReduction::bitvec_and– the result will be the bitvector AND of the given numbers (represented as integers).
9. CkReduction::bitvec_or– the result will be the bitvector OR of the given numbers (represented as integers).
10. CkReduction::set– the result will be a verbatim concatenation of all the contributed data, separated into CkReduction::setElement records. The data contributed can be of any length, and can vary across array elements or reductions. To extract the data from each element, see the description below.
11. CkReduction::concat– the result will be a byte-by-byte concatentation of all the contributed data. The contributed elements are not delimiter-separated.

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

```
class CkReduction::setElement
{
public:
  int dataSize;//The length of the data array below
  char data[];//The (dataSize-long) array of data
  CkReduction::setElement *next(void);
};
```

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

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

The reduction set order is undefined. You should add a source field to the contributed elements if you need to know which array element gave a particular contribution. Additionally, if the contributed elements are of a complex data type, you will likely have to supply code for serializing/deserializing them. Consider using the PUP interface see 7 to simplify your object serialization needs.

If the outcome of your reduction is dependent on the order in which data elements are processed, or if your data is just too heterogenous to be handled elegantly by the predefined types and you don't want to undertake multiple reductions, it may be best to define your own reduction type. See the next section (Section 13.1.4) for details.

### 5.0.8 Destroying Arrays

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

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

You must ensure that no messages are sent to a deleted element. After destroying an element, you may insert a new element at its index.

# Chapter 6

# Expressing Parallel Control Flow

### 6.0.1 Structured Control Flow: Structured Dagger

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

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

```
// in .ci file
chare ComputeObject {
  entry void ComputeObject();
  entry void startStep();
  entry void firstInput(Input i);
  entry void secondInput(Input j);
};

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

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

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

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

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

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

The most basic construct in SDAG is the `atomic` block. Atomic blocks contain sequential C++ code. They're called atomic because the code within them executes without returning control to the Charm++ runtime scheduler, and thus avoiding interruption from incoming messages. The earlier example can be adapted to use atomic blocks as follows:

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

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

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

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

Note that chare classes containing SDAG code must make a few additional calls in addition to inheriting from their `CBase_Foo` class: incorporate the `Foo_SDAG_CODE` generated-code macro in the class, and call `__sdag_init()` in the class's constructor(s).

Atomic blocks can also specify a textual 'label' that will appear in traces, as follows:

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

In order to control the sequence in which entry methods are processed, SDAG provides the `when` construct. Entry methods defined by a `when` are not executed immediately when a message tergeting them is delivered, but instead are held until control flow in the chare reaches a corresponding `when` clause. Conversely, when control flow reaches a `when` clause, the generated code checks whether a corresponding message has arrived: if one has arrived, it is processed; otherwise, control is returned to the CHARM++ scheduler.

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

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

// in C++ file
class ComputeObject : public CBase_ComputeObject {
  ComputeObject_SDAG_Code
public:
  ComputeObject() {
    __sdag_init();
    startStep();
  }

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

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

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

```
entry void startStep() {
  when firstInput(Input first),
       secondInput(Input second)
    atomic {
      computeInteractions(first, second);
    }
};
```

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

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

// in C++ file
```

```
class ComputeObject : public CBase_ComputeObject {
  ComputeObject_SDAG_Code

public:
  ComputeObject() {
    __sdag_init();
    runForever();
  }

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

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

```
// in .ci file
chare ComputeObject {
  entry void ComputeObject();
  entry void runForever() {
    for(iter = 0; iter < n; ++iter) {
      when firstInput(Input first),
           secondInput(Input second) atomic {
        computeInteractions(first, second);
      }
    }
  };
  entry void firstInput(Input i);
  entry void secondInput(Input j);
};

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

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

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

Note that `int iter;` is declared in the chare's class definition and not in the `.ci` file. This is necessary because the CHARM++ interface translator does not fully parse the declarations in the `for` loop header, because of the inherent complexities of C++.

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

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

In the running example, suppose each input needed to be preprocessed

Typically atomic blocks hold the code that actually deals with incoming messages in a `when` statement, or to do local operations before a message is sent or after it's received.

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

```
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);
  }
```

Contrast the compute chare-object example in figure **??** with a thread-based implementation of the same scheme in figure **??**. 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.

```
  array[1D] compute_object {
    entry void recv_first(Patch *first);
    entry void recv_second(Patch *first);
    entry void compute_object(MSG *msg){
      atomic {
```

```
      PatchManager->Get(msg->first_index,...);
      PatchManager->Get(msg->second_index,...);
    }
    overlap {
      when recv_first(Patch *first) atomic { filter(first); }
      when recv_second(Patch *second) atomic { filter(second); }
    }
    atomic { computeInteractions(first, second); }
  }
}
```

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

**Usage**

You can use SDAG to implement entry methods for any chare, chare array, group, or nodegroup. Any entry method implemented using SDAG must be implemented in the interface (.ci) file for its class. An SDAG entry method consists of a series of SDAG constructs of the following kinds:

- `atomic` blocks: Atomic blocks simply contain sequential C++ code. They're called atomic because the code within them executes without interruption from incoming messages. Typically atomic blocks hold the code that actually deals with incoming messages in a `when` statement, or to do local operations before a message is sent or after it's received.
- `overlap` blocks:
- `when` statements: These statement, also called triggers, indicate that we expect an incoming message of a particular type, and provide code to handle that message when it arrives. They commonly occur inside of `overlap` blocks, loops, and other control flow statements.
- `forall` loops: These loops are used when each iteration of a loop can be performed in parallel. This is in contrast to a regular `for` loop, in which each iteration is executed sequentially.
- `if`, `for`, and `while` statements: these statements have the same meaning as the normal `if`, `for`, and `while` loops in sequential C++ programs. This allows the programmer to use common control flow constructs outside the context of atomic blocks.

*Structured Dagger* code can be inserted into the .ci file for any array, group, or chare's entry methods. If you've added *Structured Dagger* code to your class, you must link in the code by:

- Adding "*className*_SDAG_CODE" inside the class declaration in the .h file. This macro defines the entry points and support code used by *Structured Dagger*. Forgetting this results in a compile error (undefined sdag entry methods referenced from the .def file).

- Adding a call to the routine "__sdag_init();" from every constructor, including the migration constructor. Forgetting this results in using uninitalized data, and a horrible runtime crash.
- Adding a call to the pup routine "__sdag_pup(p);" from your pup routine. Forgetting this results in failure after migration.

For example, an array named "Foo" that uses sdag code might contain:

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

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

For more details regarding *Structured Dagger*, look at the example located in the `examples/charm++/hello/sdag` directory in the Charm++ distribution.

### Grammar

#### Tokens

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

#### 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> ")"
```

# Chapter 7

# Serialization Using the PUP Framework

The PUP framework is a generic way to describe the data in an object and to use that description for any task requiring serialization. The CHARM++ system can use this description to pack the object into a message, and unpack the message into a new object on another processor. The name thus is a contraction of the words Pack and UnPack (PUP).

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

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

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

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

### 7.0.1 PUP contract

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

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

### PUP operator

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

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

- A simple type, including char, short, int, long, float, or double. In this case, `p|a;` copies the data in-place. This is equivalent to passing the type directly to the PUP::er using `p(a)`.
- Any object with a pup method. In this case, `p|a;` calls the object's pup method. This is equivalent to the statement `a.pup(p);`.
- A pointer to a PUP::able object, as described in Section 13.1.6. In this case, `p|a;` allocates and copies the appropriate subclass.
- An object with a PUPbytes(*myClass*) declaration in the header. In this case, `p|a;` copies the object as plain bytes, like memcpy.
- An object with a custom `operator |` defined. In this case, `p|a;` calls the custom `operator |`.

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

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

### PUP STL Container Objects

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

### PUP Dynamic Data

As usual in C++, pointers and allocatable objects usually require special handling. Typically this only requires a `p.isUnpacking()` conditional block, where you perform the appropriate allocation. See Section 13.1.6 for more information and examples.

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

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

### PUP as bytes

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

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

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

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

### PUP overhead

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

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

### PUP structured dagger

Please note that if your object contains Structured Dagger code (see section 6.0.1) you must call the generated method __sdag_pup, after any superclass pup methods, to correctly pup the Structured Dagger state:

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

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

### PUP modes

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

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

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

More specialized modes and PUP::ers are described in section 13.1.6.

### 7.0.2 PUP Life Cycle

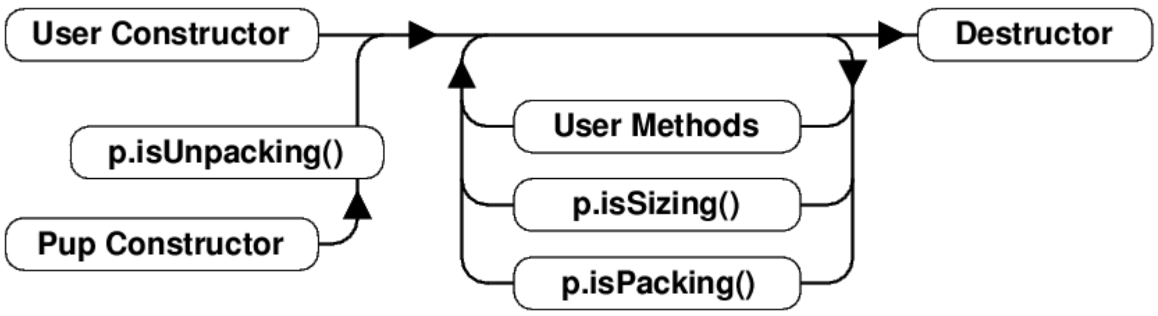

Figure 7.1: Life cycle of an object with a pup method.

The life cycle of an object with a pup method is shown in Figure 7.1. As usual in C++, objects are constructed, do some processing, and are then destroyed.

Objects can be created in one of two ways: they can be created using a normal constructor as usual; or they can be created using their pup constructor. The pup constructor for CHARM++ array elements and PUP::able objects is a "migration constructor" that takes a single "CkMigrateMessage *"; for other objects, such as parameter marshalled objects, the pup constructor has no parameters. The pup constructor is always followed by a call to the object's pup method in `isUnpacking` mode.

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

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

### 7.0.3 Migratable Array Elements using PUP

Array objects can migrate from one PE to another. For example, the load balancer (see section 8) might migrate array elements to better balance the load between processors. For an array element to be migratable, it must implement a *pup* method. The standard PUP contract (see section 7.0.1) and constraints wrt to serializing data, and use of Structured Dagger apply. A simple example for an array follows:

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

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

//In the .C file:
void A2::pup(PUP::er &p)
{
    CBase_A2::pup(p); //<- MUST call superclass's pup method
    p|nt;
    p|chr;
    p(flt,7);
    p|numDbl;
    if (p.isUnpacking()) dbl=new double[numDbl];
    p(dbl,numDbl);
}
```

### 7.0.4 Marshalling User Defined Data Types via PUP

Parameter marshalling requires serialization and is therefore implemented using the PUP framework. User defined data types passed as parameters must abide by the standard PUP contract (see section 7.0.1).

For efficiency, arrays are always copied as blocks of bytes and passed via pointers. This means classes that need their pup routines to be called, such as those with dynamically allocated data or virtual methods cannot be passed as arrays–use CkVec or STL vectors to pass lists of complicated user-defined classes. For historical reasons, pointer-accessible structures cannot appear alone in the parameter list (because they are confused with messages).

The order of marshalling operations on the send side is:

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

The order of marshalling operations on the receive side is:

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

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

# Chapter 8

# Load Balancing

Load balancing in Charm++ is enabled by its ability to place, or migrate, chares or chare array elements. Typical application usage to exploit this feature will construct many more chares than processors, and enable their runtime migration.

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

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

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

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

### Measurement-based Object Migration Strategies

In Charm++, objects (except groups, nodegroups) can migrate from processor to processor at runtime. Object migration can potentially improve the performance of the parallel program by migrating objects from overloaded processors to underloaded ones.

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

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

- Load balancing database provides the interface of almost all load balancing calls. On each processor, it stores the load balancing instrumented data and coordinates the load balancing manager and balancer. It is implemented as a Chare Group called LBDatabase.
- Load balancer or strategy takes the load balancing database and produces the new mapping of the objects. In CHARM++, it is implemented as Chare Group inherited from BaseLB. Three kinds of schemes are implemented: (a) centralized load balancers, (b) fully distributed load balancers and (c) hierarchical load balancers.

**Available Load Balancing Strategies**

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

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

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

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

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

- **RandCentLB**: Randomly assigns objects to processors;
- **MetisLB**: Uses METIS™ to partitioning object communication graph.
- **GreedyLB**: Uses a greedy algorithm that always assigns the heaviest object to the least loaded processor.
- **GreedyCommLB**: Extends the greedy algorithm to take the communication graph into account.
- **TopoCentLB**: Extends the greedy algorithm to take processor topology into account.
- **RefineLB**: Moves objects away from the most overloaded processors to reach average, limits the number of objects migrated.
- **RefineCommLB**: Same idea as in RefineLB, but takes communication into account.
- **RefineTopoLB**: Same idea as in RefineLB, but takes processor topology into account.
- **ComboCentLB**: A special load balancer that can be used to combine any number of centralized load balancers mentioned above.

Listed below are the distributed load balancers:

- **NeighborLB**: A neighborhood load balancer in which each processor tries to average out its load only among its neighbors.
- **WSLB**: A load balancer for workstation clusters, which can detect load changes on desktops (and other timeshared processors) and adjust load without interfering with other's use of the desktop.

An example of a hierarchical strategy can be found in:

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

Users can choose any load balancing strategy they think is appropriate for their application. The compiler and runtime options are described in section 8.

**Load Balancing Chare Arrays**

The load balancing framework is well integrated with chare array implementation – when a chare array is created, it automatically registers its elements with the load balancing framework. The instrumentation of compute time (WALL/CPU time) and communication pattern is done automatically and APIs are provided for users to trigger the load balancing. To use the load balancer, you must make your array elements migratable (see migration section above) and choose a load balancing strategy (see the section 8 for a description of available load balancing strategies).

There are three different ways to use load balancing for chare arrays to meet different needs of the applications. These methods are different in how and when a load balancing phase starts. The three methods are: **periodic load balancing mode**, **at sync mode** and **manual mode**.

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

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

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

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

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

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

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

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

   The more commonly used approach is to force the object to be idle until load balancing finishes. The user places an AtSync call at the end of some iteration and when all elements reach that call load balancing is triggered. The objects can start executing again when ResumeFromSync() is called. In this case, the user redefines ResumeFromSync() to trigger the next iteration of the application. This manual way of using the at sync mode results in a barrier at load balancing (see example here 8).

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

3. **Manual mode**: The load balancer can be programmed to be started manually. To switch to the manual mode, the application calls *TurnManualLBOn()* on every processor to prevent the load balancer from starting automatically. *TurnManualLBOn()* should be called as early as possible in the program. It could be called at the initialization part of the program, for example from a global variable constructor, or in an initcall(Section 16.0.3). It can also be called in the constructor of a static array and definitely before the *doneInserting* call for a dynamic array. It can be called multiple times on one processor, but only the last one takes effect.

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

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

### Migrating objects

Load balancers migrate objects automatically. For an array element to migrate, user can refer to Section 7.0.3 for how to write a "pup" for an array element.

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

An array element can migrate by calling the migrateMe(*destination processor*) member function– this call must be the last action in an element entry point. The system can also migrate array elements for load balancing (see the section 8).

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

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

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

### Other utility functions

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

- **LBTurnInstrumentOn()** and **LBTurnInstrumentOff()**: are plain C functions to control the load balancing statistics instrumentation on or off on the calling processor. No implicit broadcast or synchronization exists in these functions. Fortran interface: **FLBTURNINSTRUMENTON()** and **FLBTURNINSTRUMENTOFF()**.

- **setMigratable(CmiBool migratable)**: is a member function of array element. This function can be called in an array element constructor to tell the load balancer whether this object is migratable or not[2].

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

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

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

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

**Compiler and runtime options to use load balancing module**

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

**Note:** linking a load balancing module is different from activating it:

- link an LB module: is to link a Load Balancer module(library) at compile time. You can link against multiple LB libraries as candidates.

- activate an LB: is to actually ask the runtime to create an LB strategy and start it. You can only activate load balancers that have been linked at compile time.

Below are the descriptions about the compiler and runtime options:

1. **compile time options:**

   - *-module NeighborLB -module GreedyCommLB ...*
     links the modules NeighborLB, GreedyCommLB etc into an application, but these load balancers will remain inactive at execution time unless overridden by other runtime options.
   - *-module CommonLBs*
     links a special module CommonLBs which includes some commonly used Charm++ built-in load balancers.
   - *-balancer GreedyCommLB*
     links the load balancer GreedyCommLB and invokes it at runtime.
   - *-balancer GreedyCommLB -balancer RefineLB*
     invokes GreedyCommLB at the first load balancing step and RefineLB in all subsequent load balancing steps.
   - *-balancer ComboCentLB:GreedyLB,RefineLB*
     You can create a new combination load balancer made of multiple load balancers. In the above example, GreedyLB and RefineLB strategies are applied one after the other in each load balancing step.

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

The list of existing load balancers is given in Section 8. Note: you can have multiple -module *LB options. LB modules are linked into a program, but they are not activated automatically at runtime. Using -balancer A at compile time will activate load balancer A automatically at runtime. Having -balancer A implies -module A, so you don't have to write -module A again, although that is not invalid. Using CommonLBs is a convenient way to link against the commonly used existing load balancers. One such load balancer, called MetisLB, requires the METIS library which is located at:

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

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

2. **runtime options:**

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

   - *+balancer help*
     `displays all available balancers that have been linked in.`
   - *+balancer GreedyCommLB*
     `invokes GreedyCommLB`
   - *+balancer GreedyCommLB +balancer RefineLB*
     `invokes GreedyCommLB at the first load balancing step and RefineLB in all subsequent load balancing steps.`
   - *+balancer ComboCentLB:GreedyLB,RefineLB*
     `same as the example in the -balancer compile time option.`

   `Note: +balancer option works only if you have already linked the corresponding load balancers module at compile time. Giving +balancer with a wrong LB name will result in a runtime error. When you have used -balancer A as compile time option, you do not need to use +balancer A again to activate it at runtime. However, you can use +balancer B to override the compile time option and choose to activate B instead of A.`

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

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

4. **Other useful runtime options**

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

   - *+LBDebug {verbose level}*
     `{verbose level} can be any positive integer number. 0 is to turn off the verbose. This option asks load balancer to output load balancing information to stdout. The bigger the verbose level is, the more verbose the output is.`
   - *+LBPeriod {seconds}*
     `{Seconds} can be any float number. This option sets the minimum period time in seconds between two consecutive load balancing steps. The default value is 1 second. That is to say that a load balancing step will not happen until 1 second after the last load balancing step.`
   - *+LBSameCpus*
     `This option simply tells load balancer that all processors are of same speed. The load balancer will then skip the measurement of CPU speed at runtime.`

- *+LBObjOnly*
  `This tells load balancer to ignore processor background load when making migration decisions.`
- *+LBSyncResume*
  `After load balancing step, normally a processor can resume computation once all objects are received on that processor, even when other processors are still working on migrations. If this turns out to be a problem, that is when some processors start working on computation while the other processors are still busy migrating objects, then this option can be used to force a global barrier on all processors to make sure that processors can only resume computation after migrations are completed on all processors.`
- *+LBOff*
  `This option turns off load balancing instrumentation of both CPU and communication usage at startup time.`
- *+LBCommOff*
  `This option turns off load balancing instrumentation of communication at startup time. The instrument of CPU usage is left on.`

**Seed load balancers - load balancing Chares at creation time**

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

1. *random*
   A strategy that places seeds randomly when they are created and does no movement of seeds thereafter. This is used as the default seed load balancer.

2. *neighbor*
   A strategy which imposes a virtual topology on the processors, load exchange happens among neighbors only. The overloaded processors initiate the load balancing and send work to its neighbors when it becomes overloaded. The default topology is mesh2D, one can use command line option to choose other topology such as ring, mesh3D and dense graph.

3. *spray*
   A strategy which imposes a spanning tree organization on the processors, results in communication via global reduction among all processors to compute global average load via periodic reduction. It uses averaging of loads to determine how seeds should be distributed.

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

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

**Compile and run time options for seed load balancers**

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

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

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

**Simple Load Balancer Usage Example - Automatic with Sync LB**

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

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

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

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

---

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

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

#define MAX_WORK_CNT 50
#define LB_INTERVAL 5

/*mainchare*/
class Main : public CBase_Main
{
private:
  int count;
public:
  Main(CkArgMsg* m)
  {
    /*....Initialization....*/
    mainProxy = thisProxy;
    CProxy_LBExample arr = CProxy_LBExample::ckNew(nElements);
    arr.doWork();
  };

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

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

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

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


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

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

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

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

#include "lbexample.def.h"
```

# Chapter 9

# Processor-Aware Constructs

### 9.0.1 Group Objects

So far, we have discussed chares separately from the underlying hardware resources to which they are mapped. However, while writing lower-level libraries it is sometimes useful to be able to refer to the PE on which a chare's entry method is being executed. The `group` [1]construct provides this facility by creating a collection of chares, such that there exists a single chare (or *branch* of the group) on each PE. Each branch has its own data members. Groups have a definition syntax similar to normal chares, and they have to inherit from the system-defined class CBase_*ClassName*, where *ClassName* is the name of group's C++ class [2].

#### Group Definition

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

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

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

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

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

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

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

#### Group Creation

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

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

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

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

In the above, ckNew returns an object of type CkGroupID, which is the globally unique identifier of the corresponding instance of the group. This identifier is common to all of the group's branches and can be obtained in the group's methods from the variable thisgroup, which is a public data member of the Group superclass.

A group can also be identified through its proxy, which can be obtained in one of three ways: (a) as the inherited thisProxy data member of the class; (b) from a call to ckNew as shown below:

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

or (c) by using a group identifier to create a proxy, as shown below:

```
  // We have ‘fooGroupID’ from the above ‘ckNew’ invocation

  // Obtain a proxy to the group from its group ID
  CProxy_Foo anotherFooProxy = CProxy_Foo(fooGroupID);
```

It is possible to specify the dependence of group creations using *CkEntryOptions*. For example, in the following code, the creation of group GroupB on each PE depends on the creation of GroupA on that PE.

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

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

// Specify the dependency through a ‘CkEntryOptions’ object
CkEntryOptions opts;
opts.setGroupDepID(groupAId);

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

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

**Method Invocation on Groups**

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

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

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

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

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

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

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

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

Thus a dynamically created chare can invoke a public method of a group without knowining the PE on which it actually resides.

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

### 9.0.2 NodeGroup Objects

The *node group* construct is similar to the group construct discussed above. That is, a node group is a collection of chares that can be addressed via globally unique identifier. However, a node group has one chare per *process*, or *logical node*, rather than one chare per PE. Therefore, each logical node hosts a single branch of the node group. When an entry method of a node group is executed on one of its branches, it executes on *some* PE within the node.

#### NodeGroup Declaration

Node groups are defined in a similar way to groups. [3] For example, in the interface file, we declare:

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

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

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

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

#### Method Invocation on NodeGroups

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

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

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

[3] As with groups, older syntax allows node groups to inherit from NodeGroup instead of a specific, generated "*CBase_*" class.

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

### NodeGroups and exclusive Entry Methods

**??**

Node groups may have exclusive entry methods. The execution of an exclusive entry method invocation is mutually exclusive with those of all other exclusive entry methods invocations. That is, an exclusive entry method invocation is not executed on a logical node as long as another exclusive entry method is executing on it. More explicitly, if a method $M$ of a nodegroup $NG$ is marked exclusive, it means that while an instance of that method is being executed by a PE within a logical node, no other PE within that logical node will execute any other *exclusive* methods. However, PEs in the logical node may still execute *non-exclusive* entry method invocations. An entry method can be marked exclusive by tagging it with the exclusive attribute, as explained in § 10.0.1.

### Accessing the Local Branch of a NodeGroup

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

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

# Part II

# Advanced Usage

# Chapter 10

# Optimizing Entry Method Invocation

### 10.0.1 Entry Method Attributes

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

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

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

**threaded** entry methods are entry methods which run in their own non-preemptible threads. These entry methods may perform blocking operations, such as calls to a sync entry method, or explicitly suspending themselves.

**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 finishes execution completely. Sync methods may have return values; however, they may only return messages. Callers must run in a thread separate from the runtime scheduler, e.g. a threaded entry methods. Calls expecting a return value will receive it as the return from the proxy invocation:

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

**exclusive** entry methods exist only on NodeGroup objects, and they do not execute while other exclusive entry methods belonging to the same NodeGroup objects are executing on the same node. For example, if one exclusive method of a NodeGroup object is executing on node N, and another one is scheduled to run on the same node, the second exclusive method will wait to execute until the first one finishes.

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

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

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

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

**inline** entry methods are entry functions in which the message is delivered immediately to the recipient if it happens to reside on the same processor. Therefore, these entry methods need to be reentrant, as they could be called multiple times recursively. If the recipient reside on another processor, a regular message is sent, and inline has no effect.

**local** entry methods are equivalent to function calls: the entry method is always called immediately. This feature is available only for Groups and Chare Arrays. The user has to guarantee that the recipient chare element reside on the same processor, a failure will result in the application to abort. In contrast will all other entry methods where input parameters are marshalled into a message, local entry methods pass them direcly to the callee. This implies that the callee can modify the caller data if this is passed by pointer or reference. Furthermore, the input parameters do not require to be PUPable. Being these entry methods always called immediately, they are allowed to have a non-void return type. Nevertheless, the returned type must be a pointer.

**python** entry methods are methods which are enabled to be called from python scripts running as explained in section 17. In order to work, the object owning the method must also be declared with the keyword python.

**reductiontarget** entry methods may be used as the target of reductions, despite not taking CkReduction-Msg as an argument. See 5.0.6.

### 10.0.2 Messages

Although CHARM++supports automated parameter marshalling for entry methods, you can also manually handle the process of packing and unpacking parameters by using messages. By using messages, you can potentially improve performance by avoiding unnecessary copying

A message encapsulates all the parameters sent to an entry method. Since the parameters are already encapsulated, sending messages is often more efficient than parameter marshalling. In addition, messages are easier to queue and store on the receive side.

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

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

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

```
message MessageType;
```

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++
};
```

### Message Creation and Deletion

Messages are allocated using the C++ new operator:

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

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

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

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

do the following:

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

To allocate a message whose class declaration is:

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

do the following:

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

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

To add a priority bitfield to this message,

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

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

In Section 10.0.2 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.

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

### Messages with Variable Length Arrays

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

Such a message is declared as

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

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

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

---

**An Example**

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

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

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

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

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

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

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

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

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

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

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

No special handling is needed for deleting varsize messages.

### Message Packing

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

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

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

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

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

```
// allocate memory for varmessage so charm can keep track of memory
static void* alloc(int msgnum, size_t size, int* array, int priobits)
{
  int totalsize, first_start, second_start;
  // array is passed in when the message is allocated using new (see below).
  // size is the amount of space needed for the part of the message known
  // about at compile time.  Depending on their values, sometimes a segfault
  // will occur if memory addressing is not on 8-byte boundary, so altered
  // with ALIGN8
  first_start = ALIGN8(size);  // 8-byte align with this macro
  second_start = ALIGN8(first_start + array[0]*sizeof(int));
  totalsize = second_start + array[1]*sizeof(double);
  VarsizeMessage* newMsg =
    (VarsizeMessage*) CkAllocMsg(msgnum, totalsize, priobits);
  // make firstArray point to end of newMsg in memory
  newMsg->firstArray = (int*) ((char*)newMsg + first_start);
  // make secondArray point to after end of firstArray in memory
  newMsg->secondArray = (double*) ((char*)newMsg + second_start);

  return (void*) newMsg;
}

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

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

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

### Custom Packed Messages

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

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

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

```
message PMessage;
```

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

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

Typically, the following tasks are done in pack method:

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

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

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

Here is an example of a packed-message implementation:

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

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

// File: pgm.C
...
void*
PackedMessage::pack(PackedMessage* inmsg)
{
  int treesize = inmsg->btree.getFlattenedSize();
  int totalsize = treesize + sizeof(int);
  char *buf = (char*)CkAllocBuffer(inmsg, totalsize);
  // buf is now just raw memory to store the data structure
  int num_nodes = inmsg->btree.getNumNodes();
  memcpy(buf, &num_nodes, sizeof(int));  // copy numnodes into buffer
  buf = buf + sizeof(int);               // don't overwrite numnodes
  // copies into buffer, give size of buffer minus header
  inmsg->btree.Flatten((void*)buf, treesize);
  buf = buf - sizeof(int);              // don't lose numnodes
  delete inmsg;
  return (void*) buf;
}

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

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

#### Immediate Messages

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

### 10.0.3 Controlling Delivery Order

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

#### Queueing Strategies

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
CK_QUEUEING_LFIFO
CK_QUEUEING_LLIFO
```

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);

CkEntryOptions opts1, opts2;
opts1.setQueueing(CK_QUEUEING_FIFO);
opts2.setQueueing(CK_QUEUEING_LIFO);

chare.entry_name(arg1, arg2, opts1);
chare.entry_name(arg1, arg2, opts2);
```

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

**Prioritized Execution**

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

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

To attach a priority field to a message, one needs to set aside space in the message's buffer while allocating the message. To achieve this, the size of the priority field in bits should be specified as a placement argument to the new operator, as described in Section 10.0.2. Although the size of the priority field is specified in bits, it is always padded to an integral number of `int`s. A pointer to the priority part of the message buffer can be obtained with this call:
void *CkPriorityPtr(MsgType msg)

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

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

Integer proiorities for parameter-marshalled messages can be achieved through `CkEntryOptions::setPriority()`:

```
CkEntryOptions opts;
opts.setPriority(7);
chare.entry_name(arg1, arg2, &opts);
```

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 $.001001_{\text{binary}}$. 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 then manually set the bits using the bit-setting operations inherent to C. To achieve this, one must know the format of the bitvector, which is as follows: the bitvector is represented as an array of unsigned integers. The most significant bit of the first integer contains the first bit of the bitvector. The remaining bits of the first integer contain the next 31 bits of the bitvector. Subsequent integers contain 32 bits each. If the size of the bitvector is not a multiple of 32, then the last integer contains 0 bits for padding in the least-significant bits of the integer.

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

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

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

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

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

### Skipping the Queue

Some operations that one might want to perform are sufficiently latency-sensitive that they should never wait in line behind other messages. The CHARM++ runtime offers two attributes for entry methods, expedited and immediate, to serve these needs. For more information on these attributes, see Section 10.0.1 and the example in `charm/pgms/charm++/megatest/immediatering.C`.

# Chapter 11

# More Chare Array Features

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

**Local Access**

You can get direct access to a local array element using the proxy's ckLocal method, which returns an ordinary C++ pointer to the element if it exists on the local processor; and NULL if the element does not exist or is on another processor.

```
A1 *a=a1[i].ckLocal();
if (a==NULL) //...is remote-- send message
else //...is local-- directly use members and methods of a
```

Note that if the element migrates or is deleted, any pointers obtained with ckLocal are no longer valid. It is best, then, to either avoid ckLocal or else call ckLocal each time the element may have migrated; e.g., at the start of each entry method.

**Advanced Array Creation**

There are several ways to control the array creation process. You can adjust the map and bindings before creation, change the way the initial array elements are created, create elements explicitly during the computation, and create elements implicitly, "on demand".

You can create all your elements using any one of these methods, or create different elements using different methods. An array element has the same syntax and semantics no matter how it was created.

**Advanced Array Creation: CkArrayOptions**

The array creation method ckNew actually takes a parameter of type CkArrayOptions. This object describes several optional attributes of the new array.

The most common form of CkArrayOptions is to set the number of initial array elements. A CkArrayOptions object will be constructed automatically in this special common case. Thus the following code segments all do exactly the same thing:

```
//Implicit CkArrayOptions
  a1=CProxy_A1::ckNew(parameters,nElements);

//Explicit CkArrayOptions
  a1=CProxy_A1::ckNew(parameters,CkArrayOptions(nElements));
```

```
//Separate CkArrayOptions
  CkArrayOptions opts(nElements);
  a1=CProxy_A1::ckNew(parameters,opts);
```

Note that the "numElements" in an array element is simply the numElements passed in when the array was created. The true number of array elements may grow or shrink during the course of the computation, so numElements can become out of date. This "bulk" constructor approach should be preferred where possible, especially for large arrays. Bulk construction is handled via a broadcast which will be significantly more efficient in the number of messages required than inserting each element individually which will require one message send per element.

CkArrayOptions contains a few flags that the runtime can use to optimize handling of a given array. If the array elements will only migrate at controlled points (such as periodic load balancing with `AtASync()`), this is signalled to the runtime by calling `opts.setAnytimeMigration(false)`[1]. If all array elements will be inserted by bulk creation or by `fooArray[x].insert()` calls, signal this by calling `opts.setStaticInsertion(true)` [2].

### Advanced Array Creation: Map Object

You can use CkArrayOptions to specify a "map object" for an array. The map object is used by the array manager to determine the "home" processor of each element. The home processor is the processor responsible for maintaining the location of the element.

There is a default map object, which maps 1D array indices in a block fashion to processors, and maps other array indices based on a hash function. Some other mappings such as round-robin (RRMap) also exist, which can be used similar to custom ones described below.

A custom map object is implemented as a group which inherits from CkArrayMap and defines these virtual methods:

```
class CkArrayMap : public Group
{
public:
  //...

  //Return an ``arrayHdl'', given some information about the array
  virtual int registerArray(CkArrayIndex& numElements,CkArrayID aid);
  //Return the home processor number for this element of this array
  virtual int procNum(int arrayHdl,const CkArrayIndex &element);
}
```

For example, a simple 1D blockmapping scheme. Actual mapping is handled in the procNum function.

```
class BlockMap : public CkArrayMap
{
 public:
  BlockMap(void) {}
  BlockMap(CkMigrateMessage *m){}
  int registerArray(CkArrayIndex& numElements,CkArrayID aid) {
    return 0;
  }
  int procNum(int /*arrayHdl*/,const CkArrayIndex &idx) {
    int elem=*(int *)idx.data();
    int penum =  (elem/(32/CkNumPes()));
    return penum;
```

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

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

```
  }
};
```

Once you've instantiated a custom map object, you can use it to control the location of a new array's elements using the setMap method of the CkArrayOptions object described above. For example, if you've declared a map object named "BlockMap":

```
//Create the map group
  CProxy_BlockMap myMap=CProxy_BlockMap::ckNew();
//Make a new array using that map
  CkArrayOptions opts(nElements);
  opts.setMap(myMap);
  a1=CProxy_A1::ckNew(parameters,opts);
```

**Advanced Array Creation: Initial Elements**

The map object described above can also be used to create the initial set of array elements in a distributed fashion. An array's initial elements are created by its map object, by making a call to populateInitial on each processor.

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

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

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

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

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

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

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

**Advanced Array Creation: Bound Arrays**

You can "bind" a new array to an existing array using the bindTo method of CkArrayOptions. Bound arrays act like separate arrays in all ways except for migration– corresponding elements of bound arrays always migrate together. For example, this code creates two arrays A and B which are bound together– A[i] and B[i] will always be on the same processor.

```
//Create the first array normally
  aProxy=CProxy_A::ckNew(parameters,nElements);
//Create the second array bound to the first
  CkArrayOptions opts(nElements);
  opts.bindTo(aProxy);
  bProxy=CProxy_B::ckNew(parameters,opts);
```

An arbitrary number of arrays can be bound together– in the example above, we could create yet another array C and bind it to A or B. The result would be the same in either case– A[i], B[i], and C[i] will always be on the same processor.

There is no relationship between the types of bound arrays– it is permissible to bind arrays of different types or of the same type. It is also permissible to have different numbers of elements in the arrays, although elements of A which have no corresponding element in B obey no special semantics. Any method may be used to create the elements of any bound array.

Bound arrays are often useful if A[i] and B[i] perform different aspects of the same computation, and thus will run most efficiently if they lie on the same processor. Bound array elements are guaranteed to always be able to interact using ckLocal (see section 11), although the local pointer must be refreshed after any migration. This should be done during the pup routine. When migrated, all elements that are bound together will be created at the new processor before pup is called on any of them, ensuring that a valid local pointer to any of the bound objects can be obtained during the pup routine of any of the others.

For example, an array *Alibrary* is implemented as a library module. It implements a certain functionality by operating on a data array *dest* which is just a pointer to some user provided data. A user defined array *UserArray* is created and bound to the array *Alibrary* to take advanatage of the functionality provided by the library. When bound array element migrated, the *data* pointer in *UserArray* is re-allocated in *pup()*, thus *UserArray* is responsible to refresh the pointer *dest* stored in *Alibrary*.

```
class Alibrary: public CProxy_Alibrary {
public:
  ...
  void set_ptr(double *ptr) { dest = ptr; }
  virtual void pup(PUP::er &p);
private:
  double *dest;           // point to user data in user defined bound array
};

class UserArray: public CProxy_UserArray {
public:
  virtual void pup(PUP::er &p) {
              p|len;
              if(p.isUnpacking()) {
                data = new double[len];
                Alibrary *myfellow = AlibraryProxy(thisIndex).ckLocal();
                myfellow->set_ptr(data);    // refresh data in bound array
              }
              p(data, len);
  }
private:
  CProxy_Alibrary  AlibraryProxy;   // proxy to my bound array
  double *data;          // user allocated data pointer
  int len;
};
```

## Advanced Array Creation: Dynamic Insertion

In addition to creating initial array elements using ckNew, you can also create array elements during the computation.

You insert elements into the array by indexing the proxy and calling insert. The insert call optionally takes parameters, which are passed to the constructor; and a processor number, where the element will be created. Array elements can be inserted in any order from any processor at any time. Array elements need not be contiguous.

If using insert to create all the elements of the array, you must call CProxy_Array::doneInserting before using the array.

```
//In the .C file:
int x,y,z;
CProxy_A1 a1=CProxy_A1::ckNew();  //Creates a new, empty 1D array
for (x=...) {
   a1[x  ].insert(parameters);  //Bracket syntax
   a1(x+1).insert(parameters);  // or equivalent parenthesis syntax
}
a1.doneInserting();

CProxy_A2 a2=CProxy_A2::ckNew();   //Creates 2D array
for (x=...) for (y=...)
   a2(x,y).insert(parameters);  //Can't use brackets!
a2.doneInserting();

CProxy_A3 a3=CProxy_A3::ckNew();   //Creates 3D array
for (x=...) for (y=...) for (z=...)
   a3(x,y,z).insert(parameters);
a3.doneInserting();

CProxy_AF aF=CProxy_AF::ckNew();   //Creates user-defined index array
for (...) {
   aF[CkArrayIndexFoo(...)].insert(parameters); //Use brackets...
   aF(CkArrayIndexFoo(...)).insert(parameters); //  ...or parenthesis
}
aF.doneInserting();
```

The doneInserting call starts the reduction manager (see "Array Reductions") and load balancer (see 8)– since these objects need to know about all the array's elements, they must be started after the initial elements are inserted. You may call doneInserting multiple times, but only the first call actually does anything. You may even insert or destroy elements after a call to doneInserting, with different semantics– see the reduction manager and load balancer sections for details.

If you do not specify one, the system will choose a processor to create an array element on based on the current map object.

## Advanced Array Creation: Demand Creation

Normally, invoking an entry method on a nonexistant array element is an error. But if you add the attribute [createhere] or [createhome] to an entry method, the array manager will "demand create" a new element to handle the message.

With [createhome], the new element will be created on the home processor, which is most efficient when messages for the element may arrive from anywhere in the machine. With [createhere], the new element is created on the sending processor, which is most efficient if when messages will often be sent from that same processor.

The new element is created by calling its default (taking no parameters) constructor, which must exist and be listed in the .ci file. A single array can have a mix of demand-creation and classic entry methods; and demand-created and normally created elements.

### User-defined Array Index Type

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

For example, if you have a structure or class named "Foo", you can use a *Foo* object as an array index by defining the class:

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

Note that *Foo*'s size must be an integral number of integers– you must pad it with zero bytes if this is not the case. Also, *Foo* must be a simple class– it cannot contain pointers, have virtual functions, or require a destructor. Finally, there is a Charm++ configuration-time option called CK_ARRAYINDEX_MAXLEN which is the largest allowable number of integers in an array index. The default is 3; but you may override this to any value by passing "-DCK_ARRAYINDEX_MAXLEN=n" to the Charm++ build script as well as all user code. Larger values will increase the size of each message.

You can then declare an array indexed by *Foo* objects with

```
//in the .ci file:
array [Foo] AF { entry AF(); ... }

//in the .h file:
class AF : public CBase_AF
{ public: AF() {} ... }

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

Note that since our CkArrayIndexFoo constructor is not declared with the explicit keyword, we can equivalently write the last line as:

```
    a[f].insert();
```

When you implement your array element class, as shown above you can inherit from CBase_*ClassName*, a class templated by the index type *Foo*. In the old syntax, you could also inherit directly from ArrayElementT. The array index (an object of type *Foo*) is then accessible as "thisIndex". For example:

```
//in the .C file:
AF::AF()
{
    Foo myF=thisIndex;
    functionTakingFoo(myF);
}
```

**Array Section**

Charm++ supports the array section operation, the section operation identifies a subset of array elements from a chare array for access via a single section proxy. Charm++ also supports array sections which are a subset of array elements in multiple chare arrays of the same type 11. A special proxy for an array section can be created given a list of array indexes of elements. Multicast operations, a broadcast to all members of a section, are directly supported in array section proxy with an unoptimized direct-sending implementation. Section reduction is not directly supported by the section proxy. However, an optimized section multicast/reduction library called "CkMulticast" is provided as a separate library module, which can be plugged in as a delegation of a section proxy for performing section-based multicasts and reductions using optimized spanning trees.

For each chare array "A" declared in a ci file, a section proxy of type "CProxySection_A" is automatically generated in the decl and def header files. In order to create an array section, the user needs to provide array indexes of all the array section members through either explicit enumeration, or an index range expression. You can create an array section proxy in your application by invoking ckNew() function of the CProxySection. For example, for a 3D array:

```
  CkVec<CkArrayIndex3D> elems;    // add array indices
  for (int i=0; i<10; i++)
    for (int j=0; j<20; j+=2)
      for (int k=0; k<30; k+=2)
         elems.push_back(CkArrayIndex3D(i, j, k));
  CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, elems.getVec(), elems.size());
```

Alternatively, one can do the same thing by providing the index range [lbound:ubound:stride] for each dimension:

```
  CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, 0, 9, 1, 0, 19, 2, 0, 29, 2);
```

The above codes create a section proxy that contains array elements of [0:9, 0:19:2, 0:29:2].

For user-defined array index other than CkArrayIndex1D to CkArrayIndex6D, one needs to use the generic array index type: CkArrayIndex.

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

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

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

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

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

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

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

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

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

The CkMulticast library is implemented using delegation(Sec. 16.0.5). A special "CkMulticastMgr" Chare Group is created as a delegation for section multicast/reduction - all the messages sent by the section proxy will be passed to the local delegation branch.

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

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

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

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

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

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

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

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

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

**Array Section Reduction** Since an array element can be a member of multiple array sections, it is necessary to disambiguate between which array section reduction it is participating in each time it contributes to one. For this purpose, a data structure called "CkSectionInfo" is created by CkMulticastMgr for each array section that the array element belongs to. During a section reduction, the array element must pass the CkSectionInfo as a parameter in the contribute(). The CkSectionInfo for a section can be retrieved from a message in a multicast entry point using function call CkGetSectionInfo:

```
  CkSectionInfo cookie;

  void SayHi(HiMsg *msg)
  {
    CkGetSectionInfo(cookie, msg);     // update section cookie every time
    int data = thisIndex;
    mcastGrp->contribute(sizeof(int), &data, CkReduction::sum_int, cookie);
  }
```

Note that the cookie cannot be used as a one-time local variable in the function, the same cookie is needed for the next contribute. This is because the cookie includes some context sensive information (e.g., the reduction counter). Subsequent invocations of CkGetSectionInfo() only updates part of the data in the cookie, rather than creating a brand new one.

Similar to array reduction, to use section based reduction, a reduction client CkCallback object must be created. You may pass the client callback as an additional parameter to contribute. If different contribute calls to the same reduction operation pass different callbacks, some (unspecified, unreliable) callback will be chosen for use.

See the following example:

```
    CkCallback cb(CkIndex_myArrayType::myReductionEntry(NULL),thisProxy);
    mcastGrp->contribute(sizeof(int), &data, CkReduction::sum_int, cookie, cb);
```

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

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

As in an array reduction, users can use built-in reduction types(Section 5.0.7) or define his/her own reducer functions (Section 13.1.4).

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

In case when a multicast root migrates, the library must reconstruct the spanning tree to get optimal performance. One will get the following warning message if not doing so: "Warning: Multicast not optimized after multicast root migrated." In current implementation, user needs to initiate the rebuilding process using resetSection.

```
void Foo::pup(PUP::er & p)
    // if I am multicast root and it is unpacking
   if (ismcastroot && p.isUnpacking())
```

```
    CProxySection_Foo   fooProxy;    // proxy for the section
    CkMulticastMgr *mg = CProxy_CkMulticastMgr(mCastGrpId).ckLocalBranch();
    mg->resetSection(fooProxy);
      // you may want to reset reduction client to root
    CkCallback *cb = new CkCallback(...);
    mg->setReductionClient(mcp, cb);
```

β

**Cross Array Sections** Cross array sections contain elements from multiple arrays. Construction and use of cross array sections is similar to normal array sections with the following restrictions.

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

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

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

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

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

The resulting cross section proxy, as in the example *arrayLowProxy*, can then be used for multicasts in the same way as a normal array section.

Note: For simplicity the example has all arrays and sections of uniform size. The size of each array and the number of elements in each array within a section can all be set independently.

# Chapter 12

# Chare Inheritance and Templates

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

### 12.0.1 Chare Inheritance

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

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

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

The class definitions of both these chares should look like:

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

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

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

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

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

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

### 12.0.2 Inheritance for Messages

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

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

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

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

### 12.0.3 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

Since many modern C++ compilers[1] require that the template definitions (in *addition* to the template declarations) be available in all sources which use them, you will need to include the templated Charm definitions in your header file. That is, given a module `stlib`, in addition to having a line `#include "stlib.decl.h"` in your header file (e.g. `stlib.h`), you also need the following lines towards the end of the file:

```
#define CK_TEMPLATES_ONLY
#include "stlib.def.h"
#undef CK_TEMPLATES_ONLY
```

This has the effect of including into the header file only those declarations which relate to templates. You will *still* need to include the file `stlib.def.h` *again* in your implementation sources (i.e., `stlib.C`) in order to pick up the rest of the (non-template-related) definitions. Note that for completely template-based libraries, this means that you might need to create an implementation file `stlib.C` when you otherwise wouldn't solely for the purpose of making sure that the non-template definitions in `stlib.def.h` are included and compiled.

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>;
```

[1] 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 parts of the definitions file `*.def.h` should be included in the C++ source file.

```
  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.

# Chapter 13

## 13.1 Callbacks

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

### 13.1.1 Creating a CkCallback Object

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

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

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

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

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

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

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

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

   This function will be called on the processor where the callback was created, so *param* is allowed to point to heap-allocated data. Of course, you are required to free any storage referenced by *param*.

2. CkCallback(CkCallback::ignore) When invoked, the callback will do nothing. This can be useful if a Charm++ library requires a callback, but you don't care when it finishes, or will find out some other way.

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

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

   ```
   CkCallback myCB(CkIndex_myChare::myEntry(NULL),myChareProxy);
   ```

5. CkCallback(int ep,const CkArrayID &id) When invoked, the callback will broadcast its message to the given entry method of the given array. An array proxy will work in the place of an array id.

6. CkCallback(int ep,const CkArrayIndex &idx,const CkArrayID &id) When invoked, the callback will send its message to the given entry method of the given array element.

7. CkCallback(int ep,const CkGroupID &id) When invoked, the callback will broadcast its message to the given entry method of the given group.

8. CkCallback(int ep,int onPE,const CkGroupID &id) When invoked, the callback will send its message to the given entry method of the given group member.

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

### 13.1.2 CkCallback Invocation

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

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

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

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

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

### 13.1.3 Synchronous Execution with CkCallbackResumeThread

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In all cases a *CkCallbackResumeThread* can be used to suspend a thread only once.

*Deprecated usage*: in the past, "thread_delay" was used to retrieve the incoming message from the callback. While that is still allowed for backward compatibility, its usage is deprecated. The old usage is subject to memory leaks and dangling pointers.

### 13.1.4 Advanced Reductions

#### Reduction Clients

After the data is reduced, it is passed to a you via a callback object, as described in section 13.1. The message passed to the callback is of type CkReductionMsg. The important members of CkReductionMsg are getSize(), which returns the number of bytes of reduction data; and getData(), which returns a "void *" to the actual reduced data.

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

```
double forces[2]=get_my_forces();
// When done, broadcast the CkReductionMsg to ‘‘myReductionEntry’’
CkCallback cb(CkIndex_myArrayType::myReductionEntry(NULL), thisProxy);
contribute(2*sizeof(double), forces,CkReduction::sum_double, cb);
```

In the case of the reduced version used for synchronization purposes, the callback parameter will be the only input parameter:

```
CkCallback cb(CkIndex_myArrayType::myReductionEntry(NULL), thisProxy);
contribute(cb);
```

If no member passes a callback to contribute, the reduction will use the *default* callback. Programmers can set the default callback for an array or group using the ckSetReductionClient proxy call on processor zero, or by passing the callback to `CkArrayOptions::setReductionClient()` before creating the array, as described in section 11. Again, a CkReductionMsg message will be passed to this callback, which must delete the message when done.

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

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

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

and the actual entry point:

```
void myReductionEntry(CkReductionMsg *msg)
{
  int reducedArrSize=msg->getSize() / sizeof(double);
  double *output=(double *) msg->getData();
  for(int i=0 ; i<reducedArrSize ; i++)
  {
   // Do something with the reduction results in each output[i] array element
   .
   .
   .
  }
  delete msg;
}
```

(See pgms/charm++/RedExample for a complete example).

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

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

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

If the target of a reduction is an entry method defined by a *whem* clause in **??**SDAG]sdag, one may wish to set a reference number or tag that SDAG can use to match the resulting reduction message. To set the tag on a reduction message, the contributors can pass an additional integer argument at the end of the `contribute()` call.

### Defining a New Reduction Type

It is possible to define a new type of reduction, performing a user-defined operation on user-defined data. This is done by creating a *reduction function*, which combines separate contributions into a single combined value.

The input to a reduction function is a list of CkReductionMsgs. A CkReductionMsg is a thin wrapper around a buffer of untyped data to be reduced. The output of a reduction function is a single CkReductionMsg containing the reduced data, which you should create using the CkReductionMsg::buildNew(int nBytes,const void *data) method.

Thus every reduction function has the prototype:

```
CkReductionMsg *reductionFn(int nMsg,CkReductionMsg **msgs);
```

For example, a reduction function to add up contributions consisting of two machine `short ints` would be:

```
CkReductionMsg *sumTwoShorts(int nMsg,CkReductionMsg **msgs)
{
  //Sum starts off at zero
  short ret[2]=0,0;
  for (int i=0;i<nMsg;i++) {
    //Sanity check:
    CkAssert(msgs[i]->getSize()==2*sizeof(short));
    //Extract this message's data
    short *m=(short *)msgs[i]->getData();
    ret[0]+=m[0];
    ret[1]+=m[1];
  }
  return CkReductionMsg::buildNew(2*sizeof(short),ret);
}
```

The reduction function must be registered with CHARM++ using CkReduction::addReducer from an initnode routine (see section 16.0.3 for details on the initnode mechanism). CkReduction::addReducer returns a CkReduction::reducerType which you can later pass to contribute. Since initnode routines are executed once on every node, you can safely store the CkReduction::reducerType in a global or class-static variable. For the example above, the reduction function is registered and used in the following manner:

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

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

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

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

### 13.1.5 All-to-All

All-to-All is a frequently encountered pattern of communication in parallel programs where each processing element sends a message to every other processing element. Variations on this pattern are also common. A processing element may want to send multiple messages to the same destination over time, for example, and not every pair of processors may need to communicate. In Charm++ we classify these scenarios under a single API with the aim of improving any type of Many-to-Many communication pattern.

Note that we are currently extending support for All-to-All communication in Charm++ and so the API may change in the future.

#### MeshStreamer

MeshStreamer optimizes the case of All-to-All and Many-to-Many communication on regular 2D and 3D machine topologies. Messages sent using MeshStreamer are routed along the dimensions of the specified topology and aggregated at intermediate destinations. When using it, the first step is to create a MeshStreamer group.

```
MeshStreamer(int totalBufferCapacity, int numRows,
             int numColumns, int numPlanes,
             const CProxy_MeshStreamerClient<dtype> &clientProxy,
             int yieldFlag = 0, int progressPeriodInMs = -1);
```

The constructor takes as input a reference to a MeshStreamerClient proxy. The user should pass in the proxy for the group which will receive the data sent using MeshStreamer. To do so, this group should inherit from the MeshStreamerClient group. Note that MeshStreamer and MeshStreamerClient are templated. The templated parameter specifies the type of data units which will be communicated.

The totalBufferCapacity parameter for the MeshStreamer constructor specifies the buffering limit of the library. When the collective number of items buffered by the local instance of the group reaches the specified limit, the library sends a message along each dimension to the destination for which it has buffered the most messages.

MeshStreamer employs a virtual topology to route messages. The topology is specified by the user. When a regular mesh partition is avilable for execution, performance will be much better if the dimensions of the virtual topology submitted by the user correspond to the physical dimensions of the machine topology. The Charm++ Topology Manager can be used to produce this information for the user at run time.

The insertData function, best used when called on the local instance of the MeshStreamer group, hands over individual units of data for transmission by the library.

```
void insertData(dtype &dataItem, const int destinationPe);
```

To receive items, the user needs to define a process function, which is a pure virtual function of MeshStreamerClient.

```
virtual void process(dtype &data)=0;
```

MeshStreamer aggregates items into messages which are sent out when internal buffers fill up or periodic time limits are reached. The message arriving at the destination index of the MeshStreamer group may contain items from various group indices. The receiveCombinedData function loops over the received items and calls the process function for each item. The user may choose to redefine this function to specify an alternate message processing behavior.

```
virtual void receiveCombinedData(MeshStreamerMessage<dtype> *msg);
```

### 13.1.6 Advanced PUP

#### Dynamic Allocation

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

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

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

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

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

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

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

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

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

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

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

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

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

    }
    ~keepsNullFoo() {delete f;}
};
```

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

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

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

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

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

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

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

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

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

### Subclass allocation via PUP::able

If the class *foo* above might have been a subclass, instead of simply using *new foo* above we would have had to allocate an object of the appropriate subclass. Since determining the proper subclass and calling the appropriate constructor yourself can be difficult, the PUP framework provides a scheme for automatically determining and dynamically allocating subobjects of the appropriate type.

Your superclass must inherit from PUP::able, which provides the basic machinery used to move the class. A concrete superclass and all its concrete subclasses require these four features:

- A line declaring PUPable *className*; in the .ci file. This registers the class's constructor.
- A call to the macro PUPable_decl(*className*) in the class's declaration, in the header file. This adds a virtual method to your class to allow PUP::able to determine your class's type.
- A migration constructor—a constructor that takes CkMigrateMessage *. This is used to create the new object on the receive side, immediately before calling the new object's pup routine.

- A working, virtual pup method. You can omit this if your class has no data that needs to be packed.

An abstract superclass—a superclass that will never actually be packed—only needs to inherit from PUP::able and include a PUPable_abstract(*className*) macro in their body. For these abstract classes, the .ci file, PUPable_decl macro, and constructor are not needed.

For example, if *parent* is a concrete superclass and *child* its subclass,

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

//In the .h file:
class parent : public PUP::able {
    ... data members ...
public:
    ... other methods ...
    parent() {...}

    //PUP::able support: decl, migration constructor, and pup
    PUPable_decl(parent);
    parent(CkMigrateMessage *m) : PUP::able(m) {}
    virtual void pup(PUP::er &p) {
        PUP::able::pup(p);//Call base class
        ... pup data members as usual ...
    }
};
class child : public parent {
    ... more data members ...
public:    ... more methods, possibly virtual ...
    child() {...}

    //PUP::able support: decl, migration constructor, and pup
    PUPable_decl(child);
    child(CkMigrateMessage *m) : parent(m) {}
    virtual void pup(PUP::er &p) {
        parent::pup(p);//Call base class
        ... pup child's data members as usual ...
    }
};
```

With these declarations, then, we can automatically allocate and pup a pointer to a parent or child using the vertical bar PUP::er syntax, which on the receive side will create a new object of the appropriate type:

```
class keepsParent {
    parent *obj; //May actually point to a child class (or be NULL)
public:
    ...
    ~keepsParent() {
        delete obj;
    }
    void pup(PUP::er &p)
    {
        p|obj;
    }
};
PUPmarshall(keepsParent);
```

This will properly pack, allocate, and unpack obj whether it is actually a parent or child object. The child class can use all the usual C++ features, such as virtual functions and extra private data.

If obj is NULL when packed, it will be restored to NULL when unpacked. For example, if the nodes of a binary tree are PUP::able, one may write a recursive pup routine for the tree quite easily:

```
// In the .ci file:
    PUPable treeNode;

// In the .h file
class treeNode : public PUP::able {
    treeNode *left;//Left subtree
    treeNode *right;//Right subtree
    ... other fields ...
public:
    treeNode(treeNode *l=NULL, treeNode *r=NULL);
    ~treeNode() {delete left; delete right;}

    // The usual PUP::able support:
    PUPable_decl(treeNode);
    treeNode(CkMigrateMessage *m) : PUP::able(m) { left=right=NULL; }
    void pup(PUP::er &p) {
        PUP::able::pup(p);//Call base class
        p|left;
        p|right;
        ... pup other fields as usual ...
    }
};
```

This same implementation will also work properly even if the tree's internal nodes are actually subclasses of treeNode.

You may prefer to use the macros PUPable_def(*className*) and PUPable_reg(*className*) rather than using PUPable in the .ci file. PUPable_def provides routine definitions used by the PUP::able machinery, and should be included in exactly one source file at file scope. PUPable_reg registers this class with the runtime system, and should be executed exactly once per node during program startup.

Finally, a PUP::able superclass like *parent* above must normally be passed around via a pointer or reference, because the object might actually be some subclass like *child*. Because pointers and references cannot be passed across processors, for parameter marshalling you must use the special templated smart pointer classes CkPointer and CkReference, which only need to be listed in the .ci file.

A CkReference is a read-only reference to a PUP::able object—it is only valid for the duration of the method call. A CkPointer transfers ownership of the unmarshalled PUP::able to the method, so the pointer can be kept and the object used indefinitely.

For example, if the entry method *bar* needs a PUP::able *parent* object for in-call processing, you would use a CkReference like this:

```
// In the .ci file:
    entry void barRef(int x,CkReference<parent> p);

// In the .h file:
    void barRef(int x,parent &p) {
      // can use p here, but only during this method invocation
    }
```

If the entry method needs to keep its parameter, use a CkPointer like this:

```
// In the .ci file:
    entry void barPtr(int x,CkPointer<parent> p);

// In the .h file:
    void barPtr(int x,parent *p) {
      // can keep this pointer indefinitely, but must eventually delete it
    }
```

Both CkReference and CkPointer are read-only from the send side—unlike messages, which are consumed when sent, the same object can be passed to several parameter marshalled entry methods. In the example above, we could do:

```
   parent *p=new child;
   someProxy.barRef(x,*p);
   someProxy.barPtr(x,p); // Makes a copy of p
   delete p; // We allocated p, so we destroy it.
```

### C and Fortran bindings

C and Fortran programmers can use a limited subset of the PUP::er capability. The routines all take a handle named pup_er. The routines have the prototype:

```
void pup_type(pup_er p,type *val);
void pup_types(pup_er p,type *vals,int nVals);
```

The first call is for use with a single element; the second call is for use with an array. The supported types are char, short, int, long, uchar, ushort, uint, ulong, float, and double, which all have the usual C meanings.

A byte-packing routine

```
void pup_bytes(pup_er p,void *data,int nBytes);
```

is also provided, but its use is discouraged for cross-platform puping.

pup_isSizing, pup_isPacking, pup_isUnpacking, and pup_isDeleting calls are also available. Since C and Fortran have no destructors, you should actually deallocate all data when passed a deleting pup_er.

C and Fortran users cannot use PUP::able objects, seeking, or write custom PUP::ers. Using the C++ interface is recommended.

### Common PUP::ers

The most common PUP::ers used are PUP::sizer, PUP::toMem, and PUP::fromMem. These are sizing, packing, and unpacking PUP::ers, respectively.

PUP::sizer simply sums up the sizes of the native binary representation of the objects it is passed. PUP::toMem copies the binary representation of the objects passed into a preallocated contiguous memory buffer. PUP::fromMem copies binary data from a contiguous memory buffer into the objects passed. All three support the size method, which returns the number of bytes used by the objects seen so far.

Other common PUP::ers are PUP::toDisk, PUP::fromDisk, and PUP::xlater. The first two are simple filesystem variants of the PUP::toMem and PUP::fromMem classes; PUP::xlater translates binary data from an unpacking PUP::er into the machine's native binary format, based on a machineInfo structure that describes the format used by the source machine.

### PUP::seekBlock

It may rarely occur that you require items to be unpacked in a different order than they are packed. That is, you want a seek capability. PUP::ers support a limited form of seeking.

To begin a seek block, create a PUP::seekBlock object with your current PUP::er and the number of "sections" to create. Seek to a (0-based) section number with the seek method, and end the seeking with the endBlock method. For example, if we have two objects A and B, where A's pup depends on and affects some object B, we can pup the two with:

```
void pupAB(PUP::er &p)
{
  ... other fields ...
  PUP::seekBlock s(p,2); //2 seek sections
  if (p.isUnpacking())
  {//In this case, pup B first
    s.seek(1);
    B.pup(p);
  }
  s.seek(0);
  A.pup(p,B);

  if (!p.isUnpacking())
  {//In this case, pup B last
    s.seek(1);
    B.pup(p);
  }
  s.endBlock(); //End of seeking block
  ... other fields ...
};
```

Note that without the seek block, A's fields would be unpacked over B's memory, with disasterous consequences. The packing or sizing path must traverse the seek sections in numerical order; the unpack path may traverse them in any order. There is currently a small fixed limit of 3 on the maximum number of seek sections.

**Writing a PUP::er**

System-level programmers may occasionally find it useful to define their own PUP::er objects. The system PUP::er class is an abstract base class that funnels all incoming pup requests to a single subroutine:

```
    virtual void bytes(void *p,int n,size_t itemSize,dataType t);
```

The parameters are, in order, the field address, the number of items, the size of each item, and the type of the items. The PUP::er is allowed to use these fields in any way. However, an isSizing or isPacking PUP::er may not modify the referenced user data; while an isUnpacking PUP::er may not read the original values of the user data. If your PUP::er is not clearly packing (saving values to some format) or unpacking (restoring values), declare it as sizing PUP::er.

# Chapter 14

# More Load Balancing

### 14.0.1 Advanced Load Balancing

#### Load Balancing Simulation

The simulation feature of load balancing framework allows the users to collect information about the compute WALL/CPU time and communication of the chares during a particular run of the program and use this information later to test different load balancing strategies to see which one is suitable for the program behavior. Currently, this feature is supported only for the centralized load balancing strategies. For this, the load balancing framework accepts the following command line options:

1. *+LBDump StepStart*
   This will dump the instrument/communication data collected by the load balancing framework starting from the load balancing step *StepStart* into a file on the disk. The name of the file is given by the *+LBDumpFile* option. The first step in the program is number 0. Negative numbers will be converted to 0.

2. *+LBDumpSteps StepsNo*
   This option specifies the number of steps for which data will be dumped to disk. If omitted, default value is 1. The program will exit after StepsNo files are dumped.

3. *+LBDumpFile FileName*
   This option specifies the base name of the file into which the load balancing data is dumped. If this option is not specified, the framework uses the default file `lbdata.dat`. Since multiple steps are allowed, a number is appended to the filename in the form `Filename.#`; this applies to both dump and simulation.

4. *+LBSim StepStart*
   This option instructs the framework to do the simulation from *StepStart* step. When this option is specified, the load balancing data from the file specified in the *+LBDumpFile* option, with the addition of the step number, will be read and this data will be used for the load balancing. The program will print the results of the balancing for a number of steps given by the *+LBSimSteps* option, and then will exit.

5. *+LBSimSteps StepsNo*
   This option has the same meaning of *+LBDumpSteps*, except that apply for the simulation mode. Default value is 1.

6. *+LBSimProcs*
   This option may change the number of processors target of the load balancer strategy. It may be used to test the load balancer in conditions where some processor crashes or someone becomes available. If this number is not changed since the original run, starting from the second step file the program will print other additional information about how the simulated load differs from the real load during the

run (considering all strategies that were applied while running). This may be used to test the validity of a load balancer prediction over the reality. If the strategies used during run and simulation differ, the additional data printed may not be useful.

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

```
./charmrun pgm +p1000 +balancer RandCentLB +LBDump 2 +LBDumpSteps 4 +LBDumpFile dump.dat
```

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

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

Please note that this does not invoke any real application run. Actually "pgm" can be replaced with any generic application which calls centralized load balancer.

**Future load predictor**

When objects do not follow the assumption that the future workload will be the same as the past, the load balancer might not have the correct information to do a correct rebalancing job. To prevent this the user can provide a transition function to the load balancer to predict what will be the future workload, given the past, instrumented one. As said, the user might provide a specific class which inherits from `LBPredictorFunction` and implement the appropriate functions. Here is the abstract class:

```
class LBPredictorFunction {
public:
  int num_params;

  virtual void initialize_params(double *x);

  virtual double predict(double x, double *params) =0;
  virtual void print(double *params) PredictorPrintf("LB: unknown model");;
  virtual void function(double x, double *param, double &y, double *dyda) =0;
};
```

- `initialize_params` by default initializes the parameters randomly. If the user knows how they should be, this function can be re-implemented.
- `predict` is the function that predicts the load according to the factors in parameters. For example, if the function is $y = ax + b$, the method in the implemented class should be like:

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

- `print` is a debugging function and it can be re-implemented to have a meaningful print of the learnt model
- `function` is a function internally needed to learn the parameters, `x` and `param` are input, `y` and `dyda` are output (the computed function and all its derivatives with respect to the parameters, respectively). For the function in the example should look like:

  ```
  void function(double x, double *param, double &y, double *dyda) {
    y = predict(x, param);
    dyda[0] = x;
    dyda[1] = 1;
  }
  ```

Other than these function, the user should provide a constructor which must initialize **num_params** to the number of parameters the model has to learn. This number is the dimension of **param** and **dyda** in the previous functions. For the given example, the constructor is {`num_params = 2;`}.

If the model behind the computation is not known, the user can leave the system to use a predefined default function.

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

1. *+LBPredictorWindow size*
   This parameter will specify how many statistics steps the load balancer will keep. The greater this number is, the better the approximation of the workload will be, but more memory is required to store the intermediate information. The default is 20.

2. *+LBPredictorDelay steps*
   This will tell how many load balancer steps to wait before considering the function parameters learnt and starting to use the mode. The load balancer will collect statistics for a *+LBPredictorWindow* steps, but it will start using the model as soon as *+LBPredictorDelay* information are collected. The default is 10.

Moreover another flag can be set to enable the predictor from command line: *+LBPredictor*.
Other than the command line options, there are some methods callable from user program to modify the predictor. These methods are:

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

### Control CPU Load Statistics

CHARM++ programmers can control CPU load data in the load balancing database before a load balancing phase is started (which is the time when load balancing database is collected and used by load balancing strategies).

In an array element, the following function can be invoked to overwrite the CPU load that is measured by load balancing framework.

```
double newTiming;
setObjTime(newTiming);
```

*setObjTime()* is defined as a method of class *CkMigratable*, which is the superclass of all array elements.

The users can also retrieve the current timing that the load balancing runtime has measured for the current array element.

```
double measuredTiming;
measuredTiming = getObjTime();
```

This is useful when the users want to derive a new CPU load based on the existing one.

### Model-based Load Balancing

You can also choose to feed load balancer with their own CPU timing of each Chare based on certain computational model of the applications.

To do so, first you need to turn off automatic CPU load measurement completely by setting:

```
usesAutoMeasure = CmiFalse;
```

in array element's constructor.

Then the you need to implement the following function to the chare array classes:

```
virtual void CkMigratable::UserSetLBLoad();      // defined in base class
```

This function serves as a callback that is called on each chare object when *AtSync()* is called and ready to do load balancing. The implementation of *UserSetLBLoad()* is simply to set the current chare object's CPU load to load balancer framework. *setObjTime()* described above can be used for this.

### Writing a new load balancing strategy

Charm++ programmers can choose an existing load balancing strategy from Charm++'s built-in strategies(see 8) for the best performance based on the characteristics of their applications. However, they can also choose to write their own load balancing strategies.

The Charm++ load balancing framework provides a simple scheme to incorporate new load balancing strategies. The programmer needs to write their strategy for load balancing based on the instrumented ProcArray and ObjGraph provided by the load balancing framework. This strategy is implemented within this function:

```
void FooLB::work(LDStats *stats) {
  /** ========================= INITIALIZATION ============================= */
  ProcArray *parr = new ProcArray(stats);
  ObjGraph *ogr = new ObjGraph(stats);

  /** ============================= STRATEGY ================================ */
  /// The strategy goes here
  /// The strategy goes here
  /// The strategy goes here
  /// The strategy goes here
  /// The strategy goes here

  /** ============================== CLEANUP ================================ */
  ogr->convertDecisions(stats);
}
```

Figure 14.1 explains the two data structures available to the strategy: ProcArray and ObjGraph. Using them, the strategy should assign objects to new processors where it wants to be migrated through the setNewPe() method.

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

### Adding a load balancer to Charm++

Let us assume that we are writing a new centralized load balancer called FooLB. The next few steps explain the steps of adding the load balancer to the Charm++ build system:

1. Create files named *FooLB.ci, FooLB.h and FooLB.C* in directory of `src/ck-ldb`. One can choose to copy and rename the files GraphPartLB.* and rename the class name in those files.
2. Implement the strategy in the *FooLB* class method — **FooLB::work(LDStats* stats)** as described in the previous section.
3. Build charm for your platform (This will create the required links in the tmp directory).
4. To compile the strategy files, first add *FooLB* in the ALL_LDBS list in charm/tmp/Makefile_lb.sh. Also comment out the line containing UNCOMMON_LDBS in Makefile_lb.sh. If FooLB will require some libraries at link time, you also need to create the dependency file called libmoduleFooLB.dep. Run the script in charm/tmp, which creates the new Makefile named "Make.lb".

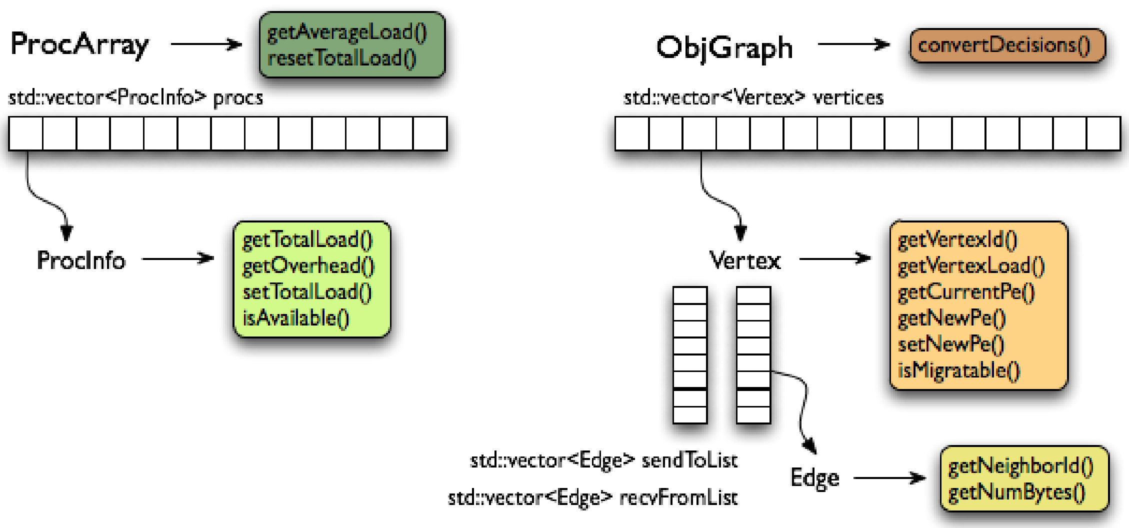

Figure 14.1: ProcArray and ObjGraph data structures to be used when writing a load balancing strategy

5. Run "make depends" to update dependence rule of Charm++ files. And run "make charm++" to compile Charm++ which includes the new load balancing strategy files.

### Understand Load Balancing Database Data Structure

To write a load balancing strategy, you need to know what information is measured during the runtime and how it is represented in the load balancing database data structure.

There are mainly 3 categories of information: a) processor information including processor speed, background load; b) object information including per object CPU/WallClock compute time and c) communication information .

The database data structure named LDStats is defined in *CentralLB.h*:

```
struct ProcStats {  // per processor
  LBRealType total_walltime;
  LBRealType total_cputime;
  LBRealType idletime;
  LBRealType bg_walltime;
  LBRealType bg_cputime;
  int pe_speed;
  double utilization;
  CmiBool available;
  int   n_objs;
}

struct LDStats { // load balancing database
  ProcStats  *procs;
  int count;

  int   n_objs;
  int   n_migrateobjs;
  LDObjData* objData;
  int   n_comm;
  LDCommData* commData;

  int  *from_proc, *to_proc;
}
```

1. *LBRealType* is the data type for load balancer measured time. It is "double" by default. User can specify the type to float at CHARM++ compile time if want. For example, ./build charm++ net-linux-x86_64 --with-lbtime-type=float;
2. *procs* array defines processor attributes and usage data for each processor;
3. *objData* array records per object information, *LDObjData* is defined in *lbdb.h*;
4. *commData* array records per communication information. *LDCommData* is defined in *lbdb.h*.

# Chapter 15

# Checkpoint/Restart

Charm++ offers a range of fault tolerance capabilities through its checkpoint/restart mechanism. Usual chare-array-based Charm++ applications, including AMPI applications, can be checkpointed to disk files and be restarted later from the files.

The basic idea behind this is straightforward: checkpointing an application is like migrating its parallel objects from the processors onto disks, and restarting is the reverse. Thanks to the migration utilities like PUP'ing(Section 7), users can decide what data to save in checkpoints and how to save them. However, unlike migration (where certain objects do not need a PUP method), checkpoint requires all the objects to implement the PUP method.

Two schemes of fault tolerance protocols are implemented.

### 15.0.1 Disk-based Checkpoint/Restart

#### Checkpointing

The API to checkpoint the application is:

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

The string *dirname* is the destination directory where the checkpoint files will be stored, and *cb* is the callback function which will be invoked after the checkpoint is done, as well as when the restart is complete. Here is an example of a typical use:

```
. . .
CkCallback cb(CkIndex_Hello::SayHi(),helloProxy);
CkStartCheckpoint("log",cb);
```

A chare array usually has a PUP routine for the sake of migration. The PUP routine is also used in the checkpointing and restarting process. Therefore, it is up to the programmer what to save and restore for the application. One illustration of this flexbility is a complicated scientific computation application with 9 matrices, 8 of which holding the intermediate results and 1 holding the final results of each timestep. To save resources, the PUP routine can well omit the 8 intermediate matrices and checkpoint the matrix with final results of each timestep.

Group and nodegroup objects(Section 9.0.1) are normally not meant to be migrated. In order to checkpoint them, however, the user has to write PUP routines for the groups and declare them as `[migratable]` in the .ci file. Some programs use *mainchares* to hold key control data like global object counts, and thus mainchares need to be checkpointed too. To do this, the programmer should write a PUP routine for the mainchare and declare them as `[migratable]` in the .ci file, just as in the case of Group and NodeGroup. In addition, the programmer also needs to put the proxy to the mainchare (usually noted as mainproxy) as a read-only data in the code, and make sure processor 0, which holds the mainchare, initiates the checkpoint.

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

**Restarting**

The user can choose to run the CHARM++ application in restart mode, i.e., restarting execution from last checkpoint. The command line option `+restart DIRNAME` is required to invoke this mode. For example:

```
  > ./charmrun hello +p4 +restart log
```

Restarting is the reverse process of checkpointing. CHARM++ allows restarting the old checkpoint on a different number of physical processors. This provides the flexibility to expand or shrink your application when the availability of computing resources changes.

Note that on restart, if the old reduction client was set to a static function, the function pointer might be lost and the user needs to register it again. A better alternative is to always use entry method of a chare object. Since all the entry methods are registered inside CHARM++ system, in the restart phase, the reduction client will be automatically restored.

After a failure, the system may contain less number of processors. Once the failed components have been repaired, some processors may become available again. Therefore, the user may need the flexibility to restart on a different number of processors than in the checkpointing phase. This is allowable by giving a different `+pN` option at runtime. One thing to note is that the new load distribution might differ from the previous one at checkpoint time, so running a load balancer (see Section 8) after restart is suggested.

If restart is not done on the same number of processors, the processor-specific data in a group/nodegroup branch cannot (and usually should not) be restored individually. A copy from processor 0 will be propagated to all the processors.

**Choosing What to Save**

In your programs, you may use chare groups for different types of purposes. For example, groups holding read-only data can avoid excessive data copying, while groups maintaining processor-specific information are used as a local manager of the processor. In the latter situation, the data is sometimes too complicated to save and restore but easy to re-compute. For the read-only data, you want to save and restore it in the PUP'er routine and leave empty the migration constructor, via which the new object is created during restart. For the easy-to-recompute type of data, we just omit the PUP'er routine and do the data reconstruction in the group's migration constructor.

A similar example is the program mentioned above, where there aree two types of chare arrays, one maintaining intermediate results while the other type holding the final result for each timestep. The programmer can take advantage of the flexibility by leaving PUP'er routine empty for intermediate objects, and do save/restore only for the important objects.

### 15.0.2 Double Memory/Disk Checkpoint/Restart

The previous disk-based fault-tolerance scheme is a very basic scheme in that when a failure occurs, the whole program gets killed and the user has to manually restart the application from the checkpoint files. The double checkpoint/restart protocol described in this subsection provides an automatic fault tolerance solution. When a failure occurs, the program can automatically detect the failure and restart from the checkpoint. Further, this fault-tolerance protocol does not rely on any reliable storage (as needed in the previous method). Instead, it stores two copies of checkpoint data to two different locations (can be memory or disk). This double checkpointing ensures the availability of one checkpoint in case the other is lost. The double in-memory checkpoint/restart scheme is useful and efficient for applications with small memory footprint at the checkpoint state. The double in-disk variant stores checkpoints into local disk, thus can be useful for applications with large memory footprint.

**Checkpointing**

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

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

where *cb* has the same meaning as in the Section 15.0.1 . Just like the above disk checkpoint described, it is up to programmer what to save. The programmer is responsible for choosing when to activate checkpointing so that the size of a global checkpoint state can be minimal.

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

```
void AMPI_MemCheckpoint()
```

### Restarting

When a processor crashes, the restart protocol will be automatically invoked to recover all objects using the last checkpoints. The program will continue to run on the surviving processors. This is based on the assumption that there are no extra processors to replace the crashed ones.

However, if there are a pool of extra processors to replace the crashed ones, the fault-tolerance protocol can also take advantage of this to grab one free processor and let the program run on the same number of processors as before the crash. In order to achieve this, CHARM++ needs to be compiled with the macro option *CK_NO_PROC_POOL* turned on.

### Double in-disk checkpoint/restart

A variant of double memory checkpoint/restart, *double in-disk checkpoint/restart*, can be applied to applications with large memory footprint. In this scheme, instead of storing checkpoints in the memory, it stores them in the local disk. The checkpoint files are named "ckpt[CkMyPe]-[idx]-XXXXX" and are stored under the /tmp directory.

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

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

# Chapter 16

# Threads, Futures, sync, barriers, quiesce, completion etc ...

### 16.0.1 Futures

Similar to Multilisp and other functional programming languages, CHARM++ provides the abstraction of *futures*. In simple terms, a *future* is a contract with the runtime system to evaluate an expression asynchronously with the calling program. This mechanism promotes the evaluation of expressions in parallel as several threads concurrently evaluate the futures created by a program.

In some ways, a future resembles lazy evaluation. Each future is assigned to a particular thread (or to a chare, in CHARM++ ) and its value will be eventually delivered to the calling program. Once the future is created, a reference is returned immediately. If the value is needed, however, the calling program blocks until the value is available.

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

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

To illustrate the use of all these functions, a Fibonacci example in CHARM++ using futures in presented below:

```
chare fib {
  entry fib(int amIroot, int n, CkFuture f);
  entry  [threaded] void run(int amIroot, int n, CkFuture f);
};

void  fib::run(int AmIRoot, int n, CkFuture f) {
   if (n< THRESHOLD)
    result =seqFib(n);
  else {
    CkFuture f1 = CkCreateFuture();
    CkFuture f2 = CkCreateFuture();
    CProxy_fib::ckNew(0,n-1,  f1);
    CProxy_fib::ckNew(0,n-2,  f2);
    ValueMsg * m1 = (ValueMsg *) CkWaitFuture(f1);
    ValueMsg * m2 = (ValueMsg *) CkWaitFuture(f2);
    result = m1->value + m2->value;
    delete m1; delete m2;
  }
  if (AmIRoot) {
    CkPrintf("The requested Fibonacci number is : %d
n", result);
    CkExit();
  } else {
    ValueMsg *m = new ValueMsg();
    m->value = result;
    CkSendToFuture(f, m);
  }
}
```

The constant *THRESHOLD* sets a limit value for computing the Fibonacci number with futures or just with the sequential procedure. Given value $n$, the program creates two futures using *CkCreateFuture*. Those futures are used to create two new chares that will carry on the computation. Next, the program blocks until the two values of the Fibonacci's recurrence have been evaluated. Function *CkWaitFuture* is used for that purpose. Finally, the program checks whether it is the root of the evaluation or not. The very first chare created with a future is going to be the root. If a chare is not a root, it must indicate its future has finished computing the value. *CkSendToFuture* is meant to return the value for the current future.

Other functions complete the API for futures. *CkReleaseFuture* destroys a future. *CkProbeFuture* test if the future has already finished computing the value of the expression.

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

### 16.0.2 Quiescence Detection

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

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

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

1. An index corresponding to the entry function that is to be called, and a handle to the chare on which that entry function should be called. The syntax of this call looks like this:

   ```
    CkStartQD(int Index,const CkChareID* chareID);
   ```

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

   myIdx=`CkIndex_`*`ChareClass`*`::`*`EntryMethod`*`(`*`parameters`*`);`

   where *ChareClass* is the C++ class of the chare containing the desired entry method, *EntryMethod* is the name of that entry method, and *parameters* are the parameters taken by the method. These parameters are only used to resolve the proper *EntryMethod*; they are otherwise ignored.

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

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

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

CkWaitQD, by contrast, does not register a callback. Rather, CkWaitQD blocks and does not return until quiescence is detected. It takes no parameters and returns no value. A call to CkWaitQD simply looks like this:

```
CkWaitQD();
```

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

### Completion Detection

Completion detection is a method for automatically detecting completion of a distributed process within an application. It is a module, and therefore is only included when "-module completion" is specified when linking your application.

Completion is reached within a distributed process when the participating objects have produced and consumed an equal number of events globally. The number of global events that will be produced and consumed does not need to be known, just the number of producers is required.

First, the detector should be constructed. This call would typically belong in application startup code (it initializes the group that keeps track of completion):

```
CProxy_CompletionDetector detector = CProxy_CompletionDetector::ckNew();
```

When it is time to start completion detection, make the following call to the library:

```
void start_detection(int num_producers, CkCallback start, CkCallback finish,
int prio_)
```

```
detector.start_detection(10, CkCallback(CkIndex_chare1::start_test(0),
                                   thisProxy),
                             CkCallback(CkIndex_chare2::finish_test(0),
                                   thisProxy), 0);
```

The `num_producers` parameter is the number of objects (chares) that will produce elements. So if every array element will produce one event, then it would be the size of the array.

The `start` callback notifies your program that it is safe to begin producing and consuming (this state is reached when the module has finished its internal initialization).

The `finish` callback is instigated when completion has been detected (all objects participating have produced and consumed an equal number of elements globally).

The `prio` parameter is the priority that the detector will run at. This is still under development, but it should be set below the application's priority if possible.

Once initialization is complete (the "start" callback is triggered), make the following call to the library:

```
void CompletionDetector::produce(int events_produced)
void CompletionDetector::produce() // 1 by default
```

```
detector.ckLocalBranch()->produce(4);
```

Once all the "events" that this chare is going to produce have been sent out, make the following call:

```
void CompletionDetector::done(int producers_done)
void CompletionDetector::done() // 1 by default
```

```
detector.ckLocalBranch()->done();
```

The application can now start consuming elements, using the following calls:

```
void CompletionDetector::consume(int events_consumed)
void CompletionDetector::consume() // 1 by default
```

```
detector.ckLocalBranch()->consume();
```

At some point, when everyone is consuming elements, completion will be reached. The system will detect that state and will trigger the `finish` callback. At that point, `start_detection` can be called again to restart the process.

### 16.0.3 initnode and initproc routines

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

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

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

Note that these routines should only do registration, not computation since Charm run-time initialization does not start yet — use a mainchare instead, which gets executed on only processor 0, to begin the computation. Initcall routines are typically used to do special registrations and global variable setup before the computation actually begins.

### 16.0.4 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 CkMyRank()

returns the rank number of the processor on which the call was made. Processing elements within a node are ranked starting from zero.

int CkMyNode()

returns the address space number (node number) on which the call was made.

int CkNumNodes()

returns the total number of address spaces.

int CkNodeFirst(int node)

returns the processor number of the first processor in this address space.

int CkNodeSize(int node)

returns the number of processors in the address space on which the call was made.

int CkNodeOf(int pe)

returns the node number on which the call was made.

int CkRankOf(int pe)

returns the rank of the given processor within its node.

The following calls provide commonly needed functions.

void CkExitAfterQuiescence()

This call informs the Charm RTS that computation on all processors should terminate as soon as the machine becomes completely idle–that is, after all messages and entry methods are finished. This is the state of quiescence, as described further in Section 16.0.2. This routine returns immediately.

double CkCpuTimer()
Returns the current value of the system timer in seconds. The system timer is started when the program begins execution. This timer measures process time (user and system).
double CkTimer()
This is an alias for either CkWallTimer on dedicated machines (such as ASCI Red) or CkCpuTimer for machines with multiple user processes per CPU (such as workstation cluster.)

### 16.0.5 Delegation

*Delegation* is a means by which a library writer can intercept messages sent via a proxy. This is typically used to construct communication libraries. A library creates a special kind of Group called a DelegationManager, which receives the messages sent via a delegated proxy.

There are two parts to the delegation interface– a very small client-side interface to enable delegation, and a more complex manager-side interface to handle the resulting redirected messages.

#### Client Interface

All proxies (Chare, Group, Array, ...) in Charm++ support the following delegation routines.
void CProxy::ckDelegate(CkGroupID delMgr);
Begin delegating messages sent via this proxy to the given delegation manager. This only affects the proxy it is called on– other proxies for the same object are *not* changed. If the proxy is already delegated, this call changes the delegation manager.
CkGroupID CProxy::ckDelegatedIdx(void) const;
Get this proxy's current delegation manager.
void CProxy::ckUndelegate(void);
Stop delegating messages sent via this proxy. This restores the proxy to normal operation.
One use of these routines might be:

```
CkGroupID mgr=somebodyElsesCommLib(...);
CProxy_foo p=...;
p.someEntry1(...); //Sent to foo normally
p.ckDelegate(mgr);
p.someEntry2(...); //Handled by mgr, not foo!
p.someEntry3(...); //Handled by mgr again
p.ckUndelegate();
p.someEntry4(...); //Back to foo
```

The client interface is very simple; but it is often not called by users directly. Often the delegate manager library needs some other initialization, so a more typical use would be:

```
CProxy_foo p=...;
p.someEntry1(...); //Sent to foo normally
startCommLib(p,...); // Calls ckDelegate on proxy
p.someEntry2(...); //Handled by library, not foo!
p.someEntry3(...); //Handled by library again
finishCommLib(p,...); // Calls ckUndelegate on proxy
p.someEntry4(...); //Back to foo
```

Sync entry methods, group and nodegroup multicast messages, and messages for virtual chares that have not yet been created are never delegated. Instead, these kinds of entry methods execute as usual, even if the proxy is delegated.

### Manager Interface

A delegation manager is a group which inherits from CkDelegateMgr and overrides certain virtual methods. Since CkDelegateMgr does not do any communication itself, it need not be mentioned in the .ci file; you can simply declare a group as usual and inherit the C++ implementation from CkDelegateMgr.

Your delegation manager will be called by CHARM++ any time a proxy delegated to it is used. Since any kind of proxy can be delegated, there are separate virtual methods for delegated Chares, Groups, NodeGroups, and Arrays.

```
class CkDelegateMgr : public Group
public:
  virtual void ChareSend(int ep,void *m,const CkChareID *c,int onPE);

  virtual void GroupSend(int ep,void *m,int onPE,CkGroupID g);
  virtual void GroupBroadcast(int ep,void *m,CkGroupID g);

  virtual void NodeGroupSend(int ep,void *m,int onNode,CkNodeGroupID g);
  virtual void NodeGroupBroadcast(int ep,void *m,CkNodeGroupID g);

  virtual void ArrayCreate(int ep,void *m,const CkArrayIndex &idx,int onPE,CkArrayID a);
  virtual void ArraySend(int ep,void *m,const CkArrayIndex &idx,CkArrayID a);
  virtual void ArrayBroadcast(int ep,void *m,CkArrayID a);
  virtual void ArraySectionSend(int ep,void *m,CkArrayID a,CkSectionID &s);
;
```

These routines are called on the send side only. They are called after parameter marshalling; but before the messages are packed. The parameters passed in have the following descriptions.

1. **ep** The entry point begin called, passed as an index into the CHARM++ entry table. This information is also stored in the message's header; it is duplicated here for convenience.
2. **m** The CHARM++ message. This is a pointer to the start of the user data; use the system routine UsrToEnv to get the corresponding envelope. The messages are not necessarily packed; be sure to use CkPackMessage.
3. **c** The destination CkChareID. This information is already stored in the message header.
4. **onPE** The destination processor number. For chare messages, this indicates the processor the chare lives on. For group messages, this indicates the destination processor. For array create messages, this indicates the desired processor.
5. **g** The destination CkGroupID. This is also stored in the message header.
6. **onNode** The destination node.
7. **idx** The destination array index. This may be looked up using the lastKnown method of the array manager, e.g., using:

   ```
   int lastPE=CProxy_CkArray(a).ckLocalBranch()->lastKnown(idx);
   ```

8. **s** The destination array section.

The CkDelegateMgr superclass implements all these methods; so you only need to implement those you wish to optimize. You can also call the superclass to do the final delivery after you've sent your messages.

# Part III

# Optional / Expert Usage

# Chapter 17

# Python scripting language

The Python scripting language in CHARM++ allows the user to dynamically execute pieces of code inside a running application, without the need to recompile. This is performed through the CCS (Converse Client Server) framework (see "Converse Manual" for more information about this). The user specifies which elements of the system will be accessible through the interface, as we will see later, and then run a client which connects to the server.

In order to exploit this functionality, Python interpreter needs to be installed into the system, and CHARM++ LIBS need to be built with:
`./build LIBS <arch> <options>`

The interface provides three different types of requests:

**Execute** requests to execute a code, it will contain the code to be executed on the server, together with the instructions on how to handle the environment;

**Print** asks the server to send back all the strings which has been printed by the script until now;

**Finished** asks the server if the current script has finished or it is still running.

There are three modes to run code on the server, ordered here by increase of functionality, and decrease of dynamic flexibility:

- **simple read/write** By implementing the read and write methods of the object exposed to python, in this way single variables may be exposed, and the code will have the possibility to modify them individually as desired. (see section 17)
- **iteration** By implementing the iterator functions in the server (see 17), the user can upload the code of a Python function and a user-defined iterator structure, and the system will apply the specified function to all the objects reflected by the iterator structure.
- **high level** By implementing python entry methods, the Python code uploaded can access them and activate complex, parallel operations that will be performed by the CHARM++ application. (see section 17)

The description will follow the client implementation first, and continuing then on the server implementation.

### The client side

In order to facilitate the interface between the client and the server, some classes are available to the user to include into the client. Currently C++ and java interfaces are provided.

C++ programs need to include `PythonCCS-client.h` into their code. This file is among the CHARM++ include files. For java, the package `charm.ccs` needs to be imported. This is located under the java directory on the CHARM++ distribution, and it provides both the Python and CCS interface classes.

There are three main classes provided: `PythonExecute`, `PythonPrint`, and `PythonFinished` which are used for the three different types of request.

All of them have two common methods to enable communication across different platforms:

**int size();** Returns the size of the class, as number of bytes that will be transmitted through the network (this includes the code and other dynamic variables in the case of `PythonExecute`).

**char *pack();** Returns a new memory location containing the data to be sent to the server, this is the data which has to be passed to the `CcsSendRequest` function. The original class will be unmodified and can be reused in subsequent calls.

A tipical invocation to send a request from the client to the server has the following format:

```
CcsSendRequest (&server, "pyCode", 0, request.size(), request.pack());
```

### PythonExecute

To execute a Python script on a running server, the client has to create an instance of `PythonExecute`, the two constructors have the following signature (java has a correspondent functionality):

```
PythonExecute(char *code, bool persistent=false, bool highlevel=false, CmiUInt4 interpreter=0);
PythonExecute(char *code, char *method, PythonIterator *info, bool persistent=false,
              bool highlevel=false, CmiUInt4 interpreter=0);
```

The second one is used for iterative requests (see 17). The only required argument is the code, a null terminated string, which will not be modified by the system. All the other parameters are optional. They refer to the possible variants that an execution request can be. In particular, this is a list of all the options present:

**iterative** If the request is a single code (false) or if it represents a function over which to iterate (true) (see 17 for more details).

**persistent** It is possible to store information on the server which will be retained across different client calls (such as simple data or complete libraries). True means that the information will be retained on the server, false means that the information will be deleted when the script terminates to run. In order to properly dispose the memory, when the last call is made (and the data is not anymore needed), this flag should be set to false. When information has been stored on the server, in order to reuse it, the interpreter field of the request should be set to the correct value (which was returned by the previous call, see later in this subsection).

**high level** In order to have the ability to call high level CHARM++ functions (available through the keyword python) this flag must be set to true. If it is false, the entire module "charm" will not be present, but the startup of the script will be faster.

**print retain** When the client decides to print some output back to the client, this data can be retrieved with a PythonPrint request. If the output is not desired, this flag can be set to false, and the output will be discarded. If it is set to true the output will be saved waiting to be retrieved by the client. The data will survive also after the termination of the Python script, and if not retrieved will waste memory on the server.

**busy waiting** Instead of returning immediately to the client a handle that can be used to retrieve prints and check if the script has finished, the server will answer to the client only when the script has terminated to run (and it will effectively work as a PythonFinished request).

These flags can be set and checked with the following routines (CmiUInt4 represent a 4 byte unsigned integer):

```
void setCode(char *set);
void setPersistent(bool set);
void setIterate(bool set);
void setHighLevel(bool set);
void setKeepPrint(bool set);
void setWait(bool set);
void setInterpreter(CmiUInt4 i);

bool isPersistent();
bool isIterate();
bool isHighLevel();
bool isKeepPrint();
bool isWait();
CmiUInt4 getInterpreter();
```

From a PythonExecute request, the server will answer with a 4 byte integer value, which is a handle for the interpreter that is running. It can be used to request for prints, check if the script has finished, and for reusing the same interpreter (if it was persistent).

A value of 0 means that there was an error and the script didn't run. This is typically due to a request to reuse of an existing interpreter which is not available, either because it was not persistent or because another script is still running on that interpreter.

#### Auto-imported modules

When a Python script is run inside a CHARM++ application, two Python modules are made available by the system. One is **ck**, the other is **charm**. The first one is always present and it represent basic functions, the second is related to high level scripting and it is present only when this is enabled (see 17 for how to enable it, and 17 for a description on how to implement charm functions).

The methods present in the `ck` module are the following:

**printstr** It accepts a string as parameter. It will write into the server stdout that string using the `CkPrintf` function call.

**printclient** It accepts a string as parameter. It will forward the string back to the client when it issues a PythonPrint request. It will buffer the strings if the `KeepPrint` option is true, otherwise it will discard them.

**mype** It requires no parameters, and it will return an integer representing the current processor where the code is executing. It is equivalent to the CHARM++ function `CkMyPe()`.

**numpes** It requires no parameters, and it will return an integer representing the total number of processors that the application is using. It is equivalent to the CHARM++ function `CkNumPes()`.

**myindex** It requires no parameters, and it will return the index of the current element inside the array, if the object under which Python is running is an array, or None if it is running under a Chare, a Group or a Nodegroup. The index will be a tuple containing as many numbers as the dimension of the array.

**read** It accepts one object parameter, and it will perform a read request to the CHARM++ object connected to the Python script, and return an object containing the data read (see 17 for a description of this functionality). An example of a call can be: value = ck.read((number, param, var2, var3))
where the double parenthesis are needed to create a single tuple object containing four values passed as a single paramter, instead of four different parameters.

**write** It accepts two object parameters, and it will perform a write request to the CHARM++ object connected to the Python script. For a description of this method, see 17. Again, only two objects need to be passed, so extra parenthesis may be needed to create tuples from individual values.

### Iterate mode

Sometimes some operations need to be iterated over all the elements in the system. This "iterative" functionality provides a shortcut for the client user to do this. As an example, suppose we have a system which contains particles, with their position, velocity and mass. If we implement `read` and `write` routines which allow us to access single particle attributes, we may upload a script which doubles the mass of the particles with velocity greater than 1:

```
size = ck.read((``numparticles'', 0));
for i in range(0, size):
    vel = ck.read((``velocity'', i));
    mass = ck.read((``mass'', i));
    mass = mass * 2;
    if (vel > 1): ck.write((``mass'', i), mass);
```

Instead of all these read and writes, it will be better to be able to write:

```
def increase(p):
    if (p.velocity > 1): p.mass = p.mass * 2;
```

This is what the "iterative" functionality provides. In order for this to work, the server has to implement two additional functions (see 17), and the client has to pass some more information together with the code. This information is the name of the function that has to be called (which can be defined in the "code" or have already been uploaded to a persistent interpreter), and a user defined structure which specifies over what data the function should be invoked. These values can be specified either while constructing the PythonExecute variable (see the second constructor in section 17), or with the following methods:

```
void setMethodName(char *name);
void setIterator(PythonIterator *iter);
```

As for the PythonIterator object, it has to be a class defined by the user, and the user has to insure that the same definition is present inside both the client and the server. The CHARM++ system will simply pass this structure as a void pointer. This structure needs to inherit from `PythonIterator`. It is recommended that no pointers are used inside this class, and no dynamic memory allocation. If this is the case, nothing else needs to be done.

If instead pointers and dynamic memory allocation is used, the following methods have to be reimplemented:

```
int size();
char * pack();
void unpack();
```

The first returns the size of the class/structure after being packed. The second returns a pointer to a newly allocated memory containing all the packed data, the returned memory must be compatible with the class itself, since later on this same memory a call to unpack will be performed. Finally, the third will do the work opposite to pack and fix all the pointers. This method will not return anything and is supposed to fix the pointers "inline".

### PythonPrint

In order to receive the output printed by the Python script, the client needs to send a PythonPrint request to the server. The constructor is:
PythonPrint(CmiUInt4 interpreter, bool Wait=true, bool Kill=false);

The interpreter for which the request is made is mandatory. The other parameters are optional. The wait parameter represents whether a reply will be sent back immediately to the client even if there is no output (false), or if the answer will be delayed until there is an output (true). The kill option set to true means that this is not a normal request, but a signal to unblock the latest print request which was blocking.

The returned data will be a non null-terminated string if some data is present (or if the request is blocking), or a 4 byte zero data if nothing is present. This zero reply can happen in different situations:

- If the request is non blocking and no data is available on the server;
- If a kill request is sent, the previous blocking request is squashed;
- If the Python code ends without any output and it is not persistent;
- If another print request arrives, the previous one is squashed and the second one is kept.

As for a print kill request, no data is expected to come back, so it is safe to call `CcsNoResponse(server)`.

The two options can also be dynamically set with the following methods:

```
void setWait(bool set);
bool isWait();

void setKill(bool set);
bool isKill();
```

### PythonFinished

In order to know when a Python code has finished executing, especially when using persistent interpreters, and a serialization of the scripts is needed, a PythonFinished request is available. The constructor is the following:
PythonFinished(CmiUInt4 interpreter, bool Wait=true);

The interpreter corresponds to the handle for which the request was sent, while the wait option refers to a blocking call (true), or immediate return (false).

The wait option can be dynamically modified with the two methods:

```
void setWait(bool set);
bool isWait();
```

This request will return a 4 byte integer containing the same interpreter value if the Python script has already finished, or zero if the script is still running.

### The server side

In order for a Charm++ object (chare, array, node, or nodegroup) to receive python requests, it is necessary to define it as python-compliant. This is done through the keyword python placed in square brackets before the object name in the .ci file. Some examples follow:

```
mainchare [python] main {...}
array [1D] [python] myArray {...}
group [python] myGroup {...}
```

In order to register a newly created object to receive Python scripts, the method `registerPython` of the proxy should be called. As an example, the following code creates a 10 element array myArray, and then registers it to receive scripts directed to "pycode". The argument of `registerPython` is the string that CCS will use to address the Python scripting capability of the object.

```
Cproxy_myArray localVar = CProxy_myArray::ckNew(10);
localVar.registerPython(‘‘pycode’’);
```

### Server read and write functions

As explained previously in subsection 17, some functions are automatically made available to the scripting code through the *ck* module. Two of these, **read** and **write** are only available if redefined by the object. The signatures of the two methods to redefine are:

```
PyObject* read(PyObject* where);
void write(PyObject* where, PyObject* what);
```

The read function receives as a parameter an object specifying from where the data will be read, and returns an object with the information required. The write function will receive two parameters: where the data will be written and what data, and will perform the update. All these `PyObject`s are generic, and need to be coherent with the protocol specified by the application. In order to parse the parameters, and create the value of the read, please refer to the manual "Extending and Embedding the Python Interpreter", and in particular to the functions `PyArg_ParseTuple` and `Py_BuildValue`.

### Server iterator functions

In order to use the iterative mode as explained in subsection 17, it is necessary to implement two functions which will be called by the system. These two functions have the following signatures:

```
int buildIterator(PyObject*, void*);
int nextIteratorUpdate(PyObject*, PyObject*, void*);
```

The first one is called once before the first execution of the Python code, and receives two parameters. The first is a pointer to an empty PyObject to be filled with the data needed by the Python code. In order to manage this object, some utility functions are provided. They are explained in subsection 17.

The second is a void pointer containing information of what the iteration should run over. This parameter may contain any data structure, and an agreement between the client and the user object is necessary. The system treats it as a void pointer since it has no information of what user defined data it contains.

The second function (`nextIteratorUpdate`) has three parameters. The first contains the object to be filled like in `buildIterator`, but this time the object contains the PyObject which was provided for the last iteration, potentially modified by the Python function. Its content can be read with the provided routines, used to retrieve the next logical element in the iterator (with which to update the parameter itself), and possibly update the content of the data inside the CHARM++ object. The second parameter is the object returned by the last call to the Python function, and the third parameter is the same data structure passed to `buildIterator`.

Both functions return an integer which will be interpreted by the system as follows:

**1** - a new iterator in the first parameter has been provided, and the Python function should be called with it;

**0** - there are no more elements to iterate.

### Server utility functions

They are inherited when declaring an object as Python-compliant, and therefore they are available inside the object code. All of them accept a PyObject pointer where to read/write the data, a string with the name of a field, and one or two values containing the data to be read/written (note that to read the data from the PyObject, a pointer needs to be passed). The strings used to identify the fields will be the same strings that the Python script will use to access the data inside the object.

The name of the function identifies the type of Python object stored inside the PyObject container (i.e String, Int, Long, Float, Complex), while the parameter of the functions identifies the C++ object type.

```
void pythonSetString(PyObject*, char*, char*);
void pythonSetString(PyObject*, char*, char*, int);
void pythonSetInt(PyObject*, char*, long);
void pythonSetLong(PyObject*, char*, long);
void pythonSetLong(PyObject*, char*, unsigned long);
void pythonSetLong(PyObject*, char*, double);
void pythonSetFloat(PyObject*, char*, double);
void pythonSetComplex(PyObject*, char*, double, double);

void pythonGetString(PyObject*, char*, char**);
void pythonGetInt(PyObject*, char*, long*);
void pythonGetLong(PyObject*, char*, long*);
void pythonGetLong(PyObject*, char*, unsigned long*);
void pythonGetLong(PyObject*, char*, double*);
void pythonGetFloat(PyObject*, char*, double*);
void pythonGetComplex(PyObject*, char*, double*, double*);
```

To handle more complicated structures like Dictionaries, Lists or Tuples, please refer to "Python/C API Reference Manual".

#### High level scripting

When in addition to the definition of the CHARM++ object as python, an entry method is also defined as python, this entry method can be accessed directly by a Python script through the *charm* module. For example, the following definition will be accessible with the python call: result = charm.highMethod(var1, var2, var3)
It can accept any number of parameters (even complex like tuples or dictionaries), and it can return an object as complex as needed.

The method must have the following signature:

```
entry [python] void highMethod(int handle);
```

The parameter is a handle that is passed by the system, and can be used in subsequent calls to return values to the Python code.

The arguments passed by the Python caller can be retrieved using the function:
PyObject *pythonGetArg(int handle);

which returns a PyObject. This object is a Tuple containing a vector of all parameters. It can be parsed using `PyArg_ParseTuple` to extract the single parameters.

When the CHARM++'s entry method terminates (by means of `return` or termination of the function), control is returned to the waiting Python script. Since the python entry methods execute within an user-level thread, it is possible to suspend the entry method while some computation is carried on in CHARM++. To start parallel computation, the entry method can send regular messages, as every other threaded entry method (see 13.1.2 for more information on how this can be done using CkCallbackResumeThread callbacks). The only difference with other threaded entry methods is that here the callback `CkCallbackPython` must be used instead of CkCallbackResumeThread. The more specialized CkCallbackPython callback works exactly like the other one, except that it handles correctly Python internal locks.

At the end of the computation, if a value needs to be returned to the Python script, the following special returning function has to be used:
void pythonReturn(int handle, PyObject* result);

where the second parameter is the Python object representing the returned value. The function `Py_-BuildValue` can be used to create this value. This function in itself does not terminate the entry method, but only sets the returning value for Python to read when the entry method terminates.

A characteristic of Python is that in a multithreaded environment (like the one provided in CHARM++), the running thread needs to keep a lock to prevent other threads to access any variable. When using high level scripting, and the Python script is suspended for long periods of time while waiting for the CHARM++ application to perform the required task, the Python internal locks are automatically released and re-acquired by the `CkCallbackPython` class when it suspends.

# Chapter 18

# Control Point Automatic Tuning Framework

β

CHARM++ currently includes an experimental automatic tuning framework that can dynamically adapt a program at runtime to improve its performance. The program provides a set of tunable knobs that are adjusted automatically by the tuning framework. The user program also provides information about the control points so that intelligent tuning choices can be made. This information will be used to steer the program instead of requiring the tuning framework to blindly search the possible program configurations.

**Warning: this is still an experimental feature not meant for production applications**

### 18.0.1 Exposing Control Points in a Charm++ Program

The program should include a header file before any of its `*.decl.h` files:

```
#include <controlPoints.h>
```

The control point framework initializes itself, so no changes need to be made at startup in the program.

The program will request the values for each control point on PE 0. Control point values are non-negative integers:

```
my_var = controlPoint("any_name", 5, 10);
my_var2 = controlPoint("another_name", 100,101);
```

To specify information about the effects of each control point, make calls such as these once on PE 0 before accessing any control point values:

```
ControlPoint::EffectDecrease::Granularity("num_chare_rows");
ControlPoint::EffectDecrease::Granularity("num_chare_cols");
ControlPoint::EffectIncrease::NumMessages("num_chare_rows");
ControlPoint::EffectIncrease::NumMessages("num_chare_cols");
ControlPoint::EffectDecrease::MessageSizes("num_chare_rows");
ControlPoint::EffectDecrease::MessageSizes("num_chare_cols");
ControlPoint::EffectIncrease::Concurrency("num_chare_rows");
ControlPoint::EffectIncrease::Concurrency("num_chare_cols");
ControlPoint::EffectIncrease::NumComputeObjects("num_chare_rows");
ControlPoint::EffectIncrease::NumComputeObjects("num_chare_cols");
```

For a complete list of these functions, see `cp_effects.h` in `charm/include`.

The program, of course, has to adapt its behavior to use these new control point values. There are two ways for a the control point values to change over time. The program can request that a new phase (with its own control point values) be used whenever it wants, or the control point framework can automatically advance to a new phase periodically. The structure of the program will be slightly different in these to cases. Sections 18.0.1 and 18.0.1 describe the additional changes to the program that should be made for each case.

**Control Point Framework Advances Phases**

The program provides a callback to the control point framework in a manner such as this:

```
    // Once early on in program, create a callback, and register it
    CkCallback cb(CkIndex_Main::granularityChange(NULL),thisProxy);
    registerCPChangeCallback(cb, true);
```

In the callback or after the callback has executed, the program should request the new control point values on PE 0, and adapt its behavior appropriately.

Alternatively, the program can specify that it wants to call `gotoNextPhase();` itself when it is ready. Perhaps the program wishes to delay its adaptation for a while. To do this, it specifies `false` as the final parameter to `registerCPChangeCallback` as follows:

```
   registerCPChangeCallback(cb, false);
```

**Program Advances Phases**

```
     registerControlPointTiming(duration); // called after each program iteration on PE 0
     gotoNextPhase(); // called after some number of iterations on PE 0
    // Then request new control point values
```

### 18.0.2 Linking With The Control Point Framework

The control point tuning framework is now an integral part of the Charm++ runtime system. It does not need to be linked in to an application in any special way. It contains the framework code responsible for recording information about the running program as well as adjust the control point values. The trace module will enable measurements to be gathered including information about utilization, idle time, and memory usage.

### 18.0.3 Runtime Command Line Arguments

Various following command line arguments will affect the behavior of the program when running with the control point framework. As this is an experimental framework, these are subject to change.

The scheme used for tuning can be selected at runtime by the use of one of the following options:

| | |
|---|---|
| `+CPSchemeRandom` | Randomly Select Control Point Values |
| `+CPExhaustiveSearch` | Exhaustive Search of Control Point Values |
| `+CPSimulAnneal` | Simulated Annealing Search of Control Point Values |
| `+CPCriticalPathPrio` | Use Critical Path to adapt Control Point Values |
| `+CPBestKnown` | Use BestKnown Timing for Control Point Values |
| `+CPSteering` | Use Steering to adjust Control Point Values |
| `+CPMemoryAware` | Adjust control points to approach available memory |

To intelligently tune or steer an application's performance, performance measurements ought to be used. Some of the schemes above require that memory footprint statistics and utilization statistics be gathered. All measurements are performed by a tracing module that has some overheads, so is not enabled by default. To use any type of measurement based steering scheme, it is necessary to add a runtime command line argument to the user program to enable the tracing module:

```
    +CPEnableMeasurements
```

The following flags enable the gathering of the different types of available measurements.

| | |
|---|---|
| `+CPGatherAll` | Gather all types of measurements for each phase |
| `+CPGatherMemoryUsage` | Gather memory usage after each phase |
| `+CPGatherUtilization` | Gather utilization & Idle time after each phase |

The control point framework will periodically adapt the control point values. The following command line flag determines the frequency at which the control point framework will attempt to adjust things.

```
  +CPSamplePeriod     number The time between Control Point Framework samples (in seconds)
```

The data from one run of the program can be saved and used in subsequent runs. The following command line arguments specify that a file named `controlPointData.txt` should be created or loaded. This file contains measurements for each phase as well as the control point values used in each phase.

```
      +CPSaveData             Save Control Point timings & configurations at completion
      +CPLoadData             Load Control Point timings & configurations at startup
  +CPDataFilename             Specify the data filename
```

It might be useful for example, to run once with `+CPSimulAnneal +CPSaveData` to try to find a good configuration for the program, and then use `+CPBestKnown +CPLoadData` for all subsequent production runs.

# Part IV

# Appendix

## .1 Further Information

### .1.1 Related Publications

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

### .1.2 Associated Tools and Libraries

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

#### Projections

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

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

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

### .1.3 Contacts

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

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

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

```
Laxmikant Kale
Department of Computer Science
University of Illinois
201 N. Goodwin Ave.
Urbana, IL 61801
```

## .2 History

The Charm software was developed as a group effort of the Parallel Programming Laboratory at the University of Illinois at Urbana-Champaign. Researchers at the Parallel Programming Laboratory keep Charm++ updated for the new machines, new programming paradigms, and for supporting and simplifying development of emerging applications for parallel processing. The earliest prototype, Chare Kernel(1.0), was developed in the late eighties. It consisted only of basic remote method invocation constructs available as a library. The second prototype, Chare Kernel(2.0), a complete re-write with major design changes. This included C language extensions to denote Chares, messages and asynchronous remote method invocation. Charm(3.0) improved on this syntax, and contained important features such as information sharing abstractions, and chare groups (called Branch Office Chares). Charm(4.0) included Charm++ and was released in fall 1993.

Charm++ in its initial version consisted of syntactic changes to C++ and employed a special translator that parsed the entire C++ code while translating the syntactic extensions. Charm(4.5) had a major change that resulted from a significant shift in the research agenda of the Parallel Programming Laboratory. The message-driven runtime system code of the Charm++ was separated from the actual language implementation, resulting in an interoperable parallel runtime system called Converse. The Charm++ runtime system was retargetted on top of Converse, and popular programming paradigms such as MPI and PVM were also implemented on Converse. This allowed interoperability between these paradigms and Charm++. This release also eliminated the full-fledged Charm++ translator by replacing syntactic extensions to C++ with C++ macros, and instead contained a small language and a translator for describing the interfaces of Charm++ entities to the runtime system. This version of Charm++, which, in earlier releases was known as *Interface Translator* Charm++, is the default version of Charm++ now, and hence referred simply as **Charm++**. In early 1999, the runtime system of Charm++ was rewritten in C++. Several new features were added. The interface language underwent significant changes, and the macros that replaced the syntactic extensions in original Charm++, were replaced by natural C++ constructs. Late 1999, and early 2000 reflected several additions to Charm++, when a load balancing framework and migratable objects were added to Charm++.

## .3 Acknowledgements

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 retargetted at Converse. Syntactic extensions in Charm++ were dropped, and a simple interface translator was developed (by Sanjeev Krishnan and Jay DeSouza) that, along with the Charm++ runtime, became the Charm++ language. Charm (5.4R1) included the following: a complete rewrite of the Charm++ runtime system (using C++) and the interface translator (done by Milind Bhandarkar), several new features such as Chare Arrays (developed by Robert Brunner and Orion Lawlor), various libraries (written by Terry Wilmarth, Gengbin Zheng, Laxmikant Kale, Zehra Sura, Milind Bhandarkar, Robert Brunner, and Krishnan Varadarajan.) A coordination language "Structured Dagger" was been implemented on top of Charm++ (Milind Bhandarkar), dynamic seed-based load balancing (Terry Wilmarth and Joshua Yelon), a client-server interface for Converse programs, and debugging support by Parthasarathy Ramachandran, Jeff Wright, and Milind Bhandarkar, Projections, the performance visualization and analysis tool, was redesigned and rewritten using Java by Michael Denardo. The test suite for Charm++ was developed by Michael Lang, Jackie Wang, and Fang Hu. Converse was been ported to ASCI Red (Joshua Yelon), Cray T3E (Robert Brunner), and SGI Origin2000 (Milind Bhandarkar). For the current version Charm 6.0 (R1), Converse has been ported to new platforms including BlueGene/[LP] (Kumar, Huang, Bhatele), Cray XT3/4 (Zheng), Apple G5, Myrinet (Zheng), and Infiniband (Chakravorty). Charm 6.0 introduces a dedicated no network SMP multicore Converse layer for stand-alone workstation experimenters (Zheng, Chakravorty, Kale, Jetley). Charm 6.0 also includes cross platform network topology aware chare placement for 3D tori and mesh networks (Kumar, Huang, Bhatele, Bohm). The test suite was extended for automated testing on all supported platforms by Gengbin Zheng. The Projection tool was substantially improved by Chee Wai Lee and Isaac Dooley. The Control Point performance tuning framework was created by Isaac Dooley. Debugging support was enhanced with memory inspection features by Filippo Gioachin. The Charisma orchestration language was implemented on top of Charm++ by Chao Huang and Sanjay Kale. Sanjay Kale, Orion Lawlor, Gengbin Zheng, Terry Wilmarth, Filippo Gioachin, Sayantan Chakravorty, Chao Huang, David

Kunzman, Isaac Dooley, Eric Bohm, Sameer Kumar, Chao Mei, Pritish Jetley, and Abhinav Bhatele, have been responsible for the changes to the system since the last release.