Chapter 2

Simple Programs and Basic Chares

In this chapter, you will learn about the basic primitives in the Charm++ language using a series of simple examples. You will learn finbout the basic parallel objects of Charm++, called *chares*, and you will also learn the basic structure of a Charm++ program.

2.1 Chares: Message-driven Objects

We provide a series of examples that illustrate the use of singleton chares, which are a good way of introducing several basic concepts in Charm++ (although most real computational science and engineering applications will be written using *chare arrays*, which we will introduce in the next chapter).

A *chare* is essentially a C++ object, with a few special properties:

- A chare class inherits from a system-defined base class.
- A chare class supports an operator for creating new instances (objects) on remote processors.
- A chare class may have a new category of methods called *entry methods*. Entry methods can be asynchronously invoked from remote processors.

It is not necessary to understand these properties completely at this point. We will go through a series of examples that will introduce and clarify these properties.

Although Charm++ is a distinct parallel programming paradigm, it is not a new language. Your programs should be written in standard C++. However, to support its parallel features, the Charm++ system needs some additional information from you, the programmer. This information is provided in an *interface file*, which has the extension `.ci`. We will learn about the contents of this file in the examples of this chapter.

Learning by Example

The best way to learn the material of this chapter is to install Charm++ on your desktop, type in the example codes, and run them.

Let us tell you what we think will be the best way to learn the material of this chapter: Obtain access to an installation of Charm++. The simplest approach would be to install a multicore version on your multicore desktop. (What? You don't yet have one? Tsk tsk.) The examples in this chapter were tested on an 8-core desktop. Alternatively, you could use a cluster version. Let's assume your friendly system administrator (which might be you yourself) has already installed Charm++. You can enter each example program into files in their own directory, compile those file and run the resulting executable. You could also cut-and-paste or just use the examples from the Charm++ website. However, we find that typing each statement by hand facilitates the better absorption of the concepts. Your style may vary.

Please note, the examples in this chapter (and many in the next chapter as well) are very basic. You will not be running anything in parallel until the last few examples, and those programs don't do anything very useful even when they are run in parallel. But we promise that these simple programs will help you to learn and remember the basic concepts. In later chapters we will get a chance to read and write several much more interesting programs.

2.1.1 Example: Hello World

The execution of every Charm++ program begins with the creation of an instance of a specially designated class called the *main* class.[1] In interface files, this class is designated by the keyword `mainchare`.

Our first program is a simple "Hello world" program that introduces many elements common to all Charm++ programs. This program has just one class called `Main`. This class happens to be a chare class. In particular, it happens to be a chare class designated as a main chare. Since we have a chare class in the program, we must tell the system about it in the interface file. The interface file for the `Main` chare is shown in figure 2.2.

A Charm++ program is organized as a collection of modules. Each module may contain one or more chare classes, and has one interface file associated with it. A module that contains a main chare is designated by the keyword `mainmodule`. Other modules are designated by the keyword `module`. We choose to name our module `hello`. This happens to be the same as the name of the file (`hello.ci`) but this is not necessary. Declarations of all the chares in each module are enclosed in the module statement:

```
module <modulename> { ... chare definitions };
```

Don't forget the ";" after the last curly brace, or else you will get some obscure error message while "compiling" the interface file.

[1] In fact there can be multiple classes designated as *main*: the system will create one instance of each main class.

For our example program, there is only one chare class in module hello. Since this happens to be a main chare, we designate it as such by the keyword mainchare. This chare has only method, namely its constructor. However, all the constructors of a chare class are *entry methods*, by definition. This is because an instance of a chare class (simply called a chare from now on), can be created from a remote processor; any time a method can be remotely invoked, it must be an entry method. Since all entry methods must be declared in the interface file, we declare the Main method here. All main chares have a standard constructor that takes a pointer to a system defined class (called CkArgMsg) as its only parameter. So, we declare that fact in our interface file. Again, don't forget the ";" after the class definition.

The hello.C file is shown in Figure 2.3. Several features in this file deserve explanation. First, notice that the class Main is defined as a subclass of CBase_Main. Where did this class come from? The Charm++ system compiles the .ci interface file, and produces some files that declare and define several additional classes for each chare class defined in that interface file. CBase_Main is such a class. Inheriting from it allows our Main class to be created as a chare class, and is thus required. The declaration of CBase_Main (and several other declarations) are stored in a file called hello.decl.h by the interface translator included with the Charm++ system. The *definitions* of CBase_Main (and other classes) are generated in a file called hello.def.h. That is why our hello.C includes those two files at its beginning and at its end, respectively.

The other difference from a regular C++ program is the absence of the main function. Instead, program execution begins with the constructor of the main chare Main. This constructor, in this program, includes only two statements. The first prints "Hello World!" to the output stream ckout. Note that the standard output stream cout does not work in a parallel program. Instead, the Charm++ runtime system (RTS) defines its own output stream named ckout.

The second statement calls CkExit(), which tells the RTS to stop the execution of the program on all its processors that are running this program. If you call exit() instead, only the processor which happens to be executing that statement will quit, without performing the requisite cleanup functions of the RTS, and the other processors will simply hang waiting for something to happen.

To compile this program, you first issue a command to translate the interface file. A script called charmc can handle such processing automatically. Thus you simply type charmc hello.ci at the command line. (At this time, charmc must be in your path, or you must specify the full path to it. See the Charm++ installation instructions if Charm++ is not already installed on your computer.)

The charmc script processes the interface file to create two files, the hello.decl.h file and the hello.def.h file, which are to be included in the .C file. Refer to figure 2.1.

You now compile the .C file. The charmc script knows (among other things) where the system header files and libraries are located. So, instead of using the normal C++ compilation

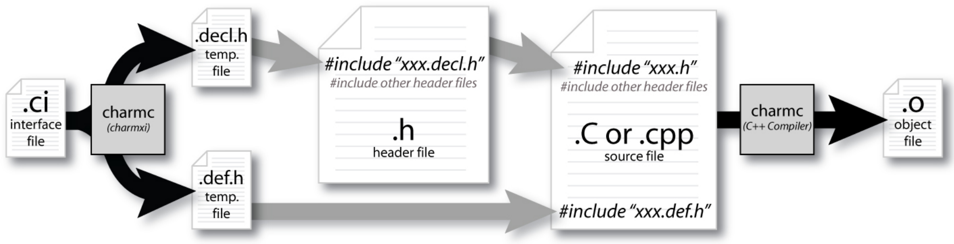

Figure 2.1: Build process of a chare class. *NOTE: If a header file isn't used for the chare class, as is the case with the examples in this chapter, simply include the xxx.decl.h file at the top of the source code file in place of the xxx.h file.*

command (such as `gcc` or `CC`), you use `charmc` to compile and link your program. `charmc` will call your normal compiler (set at installation time internally).

```
charmc hello.ci
charmc -c hello.C
charmc -o hello hello.C
```

At this point, you have created an executable file `hello`. How to execute this in parallel depends on what machine you are using.

On a multicore (or even a single core) desktop, you type:

```
hello +p7
```

The `+p7` tells the system to use seven cores[2].

On a cluster of workstations connected by ethernet, you type:

```
charmrun hello +p7
```

The `charmrun` script, which is created in your folder[3] by `charmc` when it creates the executable, is responsible for starting processes on appropriate workstations, along with other startup and monitoring functions.

If everything worked correctly, you should see the words "Hello World!" on your screen when you run the program. Otherwise, this a good point at which to get some help and make sure your installation works correctly in the context of this simple example.

[2]It actually fires seven Posix threads on Linux machines.

[3]We will use the words "folder" and "directory" interchangeably.

```
mainmodule hello {

  mainchare Main {
    entry Main(CkArgMsg *m);
  };
};
```

Figure 2.2: Hello World: the interface file `hello.ci`

```
#include <stdio.h>
#include "hello.decl.h"


/*mainchare*/
class Main : public CBase_Main
{
public:
  Main(CkArgMsg* m)
  {
      ckout << "Hello World!" << endl;
      CkExit();
  };
};

#include "hello.def.h"
```

Figure 2.3: Hello World: the C++ file `hello.C`

2.1.2 Example: Hello World with Command Line Arguments

Remember the argument `CkArgMsg *m` for the constructor of the main chare, in the example above? `CkArgMsg` is a class with two members: `argc`, which is an integer, and `argv`, which is a pointer to an array of strings. This is how command line arguments are handed over to the application program: `argc` is the number of command line arguments, and `argv[i]` is the string representing the i^{th} command line argument.

It is important to note that some of the command line arguments are "consumed" by the Charm++ runtime, and are not passed to the main chare. In particular, the `+p7` argument is not passed to the main chare. As usual, the `argv[0]` contains the name of the executable, which in this case is `hello`.

```
 1  #include <stdio.h>
 2  #include "hello.decl.h"
 3
 4
 5  /*mainchare*/
 6  class Main : public CBase_Main
 7  {
 8  public:
 9    Main(CkArgMsg* m)
10    {
11        ckout << "Hello World!" << endl;
12        if (m->argc > 1)
13            ckout << "and Hello " << m->argv[1] << "!!!" << endl;
14        CkExit();
15    };
16  };
17
18  #include "hello.def.h"
```

Figure 2.4: Processing Command Line arguments: the file `hello.C`

To illustrate this further, consider a small variation of the above program, as shown in Figure 2.4. Note that the only difference is the addition of lines 12 and 13, which print the string `argv[1]`. Try running this program with an additional argument, which could be your first name:

```
> charmrun hello Sanjay +p7

Hello World!
```

```
and Hello Sanjay!!!
```

2.2 Creating Chares

Since the execution of the program begins with the creation of one instance of the main chare, it is clear that the main chare must create at least some of the other chare objects (which we will simply call *chares* from now on) if the program computation is to have multiple chares. Either the main chare creates all the chares of the program, or it creates chares that create other chares some time during their execution, and so on. Our next example shows how chares are created.

Figure 2.5 shows the interface file for this program. This file specifies that in addition to the mainchare `Main` (which still has only one method, a constructor, in this example), we now have another chare class named `Simple`, which also has only a constructor entry method. It takes an integer and a double precision floating point number as its parameters.

The correspnding `C++` file is shown in Figure 2.6. Since chares are special objects, which may live on different processors than the one that created them, the standard `new` call will not do. If you were to call `new(12, pi)`, all that you would get is the creation of an instance of the `Simple` chare on the same processor that the main chare is running on, and it will not be registered with the runtime system, so no messages can be sent to it. It would be just a sequential `C++` object. Instead, one uses the `ckNew` class method of the system-generated class `CProxy_Simple`. This is one of the classes generated by the system using your interface file, and its declatation is included in `MyModule.decl.h`.

```
mainmodule MyModule {

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

  chare Simple {
    entry Simple(int x, double y);
  };

};
```

Figure 2.5: Creating Chares: the interface file `hello.ci`

Incidentally, note that the name of this file is not `hello.decl.h`: the names of generated files are `<Modulename>.decl.h` and `<Modulename>.def.h` respectively, where `<Modulename>`

```
#include <stdio.h>
#include "MyModule.decl.h"

/*mainchare*/
class Main : public CBase_Main
{
public:
  Main(CkArgMsg* m)
  {
      ckout << "Hello World!" << endl;
      if (m->argc > 1)
          ckout << "and Hello " << m->argv[1] << "!!!" << endl;
      double pi = 3.1415;
      CProxy_Simple::ckNew(12, pi);
  };
};

class Simple : public CBase_Simple
{
public:
 Simple(int x, double y)
   {
       ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
       ckout << "Area of a circle of radius" << x << " is " << y*x*x << endl;
       CkExit();
   }
};

#include "MyModule.def.h"
```

Figure 2.6: Creating Chares: the C++ file `hello.C`

is what you specified inside the interface file.

The `ckNew` call tells the Charm++ RTS to create, at its convenience some time in future, an instance of the `Simple` chare on some processor of its choosing, with the parameters specified.

The constructor of the `Simple` chare itself just prints two lines, with the second line printing the area of a circle. The two parameters passed to it are the radius of the circle and the value of π. Of course, since value of π is constant, you shouldn't have to pass it, but passing it as a parameter here serves our purpose of illustrating parameter passing.

One thing in particular should be noted about print statements. Since different chares may be running on different processors, and their output may be merged depending on what arrives first at the processor in charge of printing to the screen, you cannot assume any order among the print statements beyond the fact that two strings printed from the same chare will appear in the same order in the output. Thus, the string coming from line 10 will appear before that from line 12, and that coming from line 23 before that of 24. But other than that, no ordering can be inferred *even though the constructor of* `Simple` *is clearly going to execute after the prints in the* `Main` *constructor*[4].

Finally, notice that we moved the `CkExit()` call into the constructor of `Simple`, since that is the last entry method that will execute.

Again, you should compile and run this program. It is simple, but it will help you get in the habit of testing things, and typing the program will reinforce what you are learning.

2.3 Asynchronous Method Invocation (a.k.a. Sending Messages)

So far, our example programs show how to write interface files and create chares. The next example shows how to use asynchronous method invocation.

The interface file of Figure 2.7 is very similar to the previous one, except that we have added a new entry method called `findArea` to `Simple`. We are now going to pass `pi` only while constructing an instance of the `Simple` chare class (i.e. a `Simple` chare), and pass only the radius to `findArea` on each subsequent call. We are also passing an additional boolean parameter `done`, which is `TRUE` (or 1) when we want to tell the `Simple` chare that this is the last query for it, and that it can exit after this.

The C++ file is shown in Figure 2.8. The main thing to notice is the on line 14, we are storing the value returned by `ckNew` in a variable `sim` of type `CProxy_Simple`. This is called the *proxy* for the created `Simple` chare. `sim` is an object which has the same entry methods

[4]There are debugging options that make such causally connected prints to appear in order. Refer to the *+syncprint* command-line option Charm++manual. Adding *+syncprint* as a command-line option to your Charm++program will cause all *CkPrintf()* statements to be queued at a single location. This option synchronizes the print statements at the cost of application performance.

as the `Simple` class, yet these methods simply copy the parameters you pass into a message, put the address of the real object and the name of the method on its "envelope", and send it to the processor that holds that object.

```
mainmodule MyModule {

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

  chare Simple {
    entry Simple(double y);
    entry void findArea(int radius, bool);
  };

};
```

Figure 2.7: Async Method Invocation: the interface file `hello.ci`

That was a simple program, right? However, the program is WRONG. If you run it, it will mostly run correctly, but once in a while (on some machines) it may terminate without processing all the `findArea` requests. Do you know why? In fact, on most machines today, it will run correctly, but the program is still incorrect. The fix will be in the next example, but let us first understand why.

[In the base mode described here] Charm++ does not guarantee that two messages sent by chare A to chare B will be delivered (and executed) in the same order! So, what you think of as the "last" message (`findArea(10,1)`), in the example above, may execute before an earlier message (say `findArea(8,0)`), and terminate the program prematurely.

Why does charm not guarantee in-order delivery? There are two reasons for this: first, their is a cost in guaranteeing in-order delivery. On many machines, the raw messages may be delivered out of order anyways. Someone somewhere has to keep track of this, using sequence numbers on messages and buffering them until the right message arrives, if you want to guarantee in-order delivery. We'd rather not pay that cost unless it is necessary. When it is necessary, you can yourself do the buffering, add the sequence numbers to the message (or use an existing field, such as the first parameter in this example). Later on, we will see another notation built on top of Charm that also ensures in-order delivery. But we want to keep the base-line Charm++ flexible to allow either usage.

Secondly, there are situations where one would rather reorder messages. On each physical processor core, Charm++ RTS runs a scheduler, which picks a message (which is an asynchronous method invocation, stored in a general "envelope"), identifies the chare object

```
#include <stdio.h>
#include "MyModule.decl.h"

/*mainchare*/
class Main : public CBase_Main
{
public:
  Main(CkArgMsg* m)
  {
      ckout << "Hello World!" << endl;
      if (m->argc > 1)
          ckout << "and Hello " << m->argv[1] << "!!!" << endl;
      double pi = 3.1415;
      CProxy_Simple sim =  CProxy_Simple::ckNew(pi);
      for (int i = 1; i< 10; i++)
          sim.findArea(i, 0);
      sim.findArea(10,1);
  };
};

class Simple : public CBase_Simple
{
private:
    float y;
public:
 Simple( double pi)
   {
       y = pi;
       ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
   }

void findArea(int r, bool done)
   {
       ckout << "Area of a circle of radius" << r << " is " << y*r*r << endl;
       if (done) CkExit();
   }
};

#include "MyModule.def.h"
```

Figure 2.8: Async Method Invocation: the C file `hello.C`

indicated on the envelope, unpacks the parameters from the message and invokes the indicated method on it. As a result of this invocation, new method invocations may get enqueued on your own queue and/or on some other processor's queue. The scheduler's queue gives the RTS an opportunity to reorder messages to the benefit of the program. For example, one can prioritize certain kinds of messages (see XXXXXX chapter for this); One wants high priority messages to the same object to execute before earlier-sent low-priority messages. This is another reason for not enforcing in-order execution.

One needs to get used to thinking asynchronously, to ensure the program will work irrespective of the order in which messages get delivered. This thinking is useful in parallel programming anyways. Even if one were to guarantee in-order delivery between a pair of objects, the asynchrony exist in parallel programs in many other ways, and so it is useful to develop this asynchronous mode of thinking.

The counterpoint to this argument is that if we can limit this asynchrony to fewer places there is smaller chance of order-dependent bugs. The logical conclusion of this is a deterministic parallel language where things always execute in a specific order (and when they don't, its guranteed that they will have the same effect). Such languages are higher level languages, and they have been built on top of Charm++.

We will return to this example after we have done a few more examples. The next example will circumvent the issue by a simple design trick, whic is worth using in such situations, where you really didn't care about the order in which various "findArea" queries are execute, but you want the program to terminate only after finishing all the queries.

2.4 Replying Using a Chare ID

In the previous example, the main chare invoked the `findArea` method on the instance of the `Simple` chare *that it had created.* It therefore had a proxy to that chare, stored in the variable `sim`. How do you invoke a method on a chare that you did not create? Somehow, you must obtain a proxy to the object. The next example illustrates one way this can be done.

Proxy objects are simple objects that can be passed in messages. If the main chare were to pass *its own proxy* to the chare it creates, that chare can store this proxy, and use it to reply back to the main chare.

In the interface file shown in Figure 2.9, we have added a new method called `printArea` to the main chare. In addition, we added one more parameter to the `Simple` chare class's constructor, of type `CProxy_Main`. Also, since the main chare can now call `CkExit` after printing the last result, we don't need to pass the boolean parameter to `findArea`.

The corresponding `C++` file is shown in Figure 2.10. Incidentally, if the declaration of an entry method in the interface file does not match that in your `.C` or `.h` file, you will get errors from the compiler such as *no matching function call.*

```
mainmodule MyModule {

  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void printArea(int r, double area);
  };

  chare Simple {
  entry Simple(double y, CProxy_Main m);
  entry void findArea(int radius, bool);
};

};
```

Figure 2.9: Replying to Chares: the interface file `hello.ci`

The main chare now initializes an object-level variable `count` which it initializes to ten, the number of chares it created. The `printArea` method prints the area returned by `findArea`, and exits once it has received ten results. Notice that even if the ten results arrive out of order, the program will exit only after all ten have been printed. Notice also that we shouldn't omit the parameter `r` and assume the i^{th} result returned will be for radius i, although the `for` loop in `Main::Main` might make you think that. The messages carrying the queries and results can, in principle, take varying amounts of time and thus return in a different order.

2.5 Creating Many Chares: A First Brush with Load Balancing

The examples so far were hardly parallel; there were at most two chares. Now we will write a program that creates a large number of chares.

We will make a `Worker` chare whose constructor does the work of calculating the area, given a radius and the value of π. By the way, computing the area is really a very tiny amount of work and doesn't deserve to be done by an entire parallel object. If that bothers you, wait for the next example, where we return to this issue of grainsize control. This example is somewhat like the program of Figure 2.6 that we saw earlier. Accordingly, there is no `findArea` method now.

The interface file is shown in Figure 2.11. We have named the main chare `Master` now, but it is otherwise very similar to earlier versions, with the exception of the *readonly* declaration, which we will explain shortly. The C++ program is shown in Figure 2.12.

We are using a different method to make the proxy of the main chare available to each

```
#include <stdio.h>
#include "MyModule.decl.h"

/*mainchare*/
class Main : public CBase_Main
{
private:
    int count;
public:
  Main(CkArgMsg* m)
  {
      ckout << "Hello World!" << endl;
      double pi = 3.1415;
      CProxy_Simple sim =  CProxy_Simple::ckNew(pi, thisProxy);
      for (int i = 1; i <= 10; i++)
          sim.findArea(i, 0);
      count = 10;  // wait for 10 responses
  };

    void printArea(int r, double area) {
        ckout << "Area of a circle of radius" << r << " is " << area << endl;
        count--;
        if (count == 0) CkExit();
    }

};

class Simple : public CBase_Simple
{
private:
    float y;
    CProxy_Main mainProxy;
public:
    Simple( double pi, CProxy_Main master)
   {
       y = pi;
       mainProxy = master;
       ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
   }

void findArea(int r, bool done)
   {
       mainProxy.printArea(r, y*r*r);

   }
};

#include "MyModule.def.h"
```

Figure 2.10: Replying to Chares: the C++ file `hello.C`

```
mainmodule MyModule {

  readonly CProxy_Master mainProxy;

  mainchare Master {
    entry Master(CkArgMsg *m);
    entry void printArea(int r, double area);
  };

  chare Worker {
  entry Worker(int r, double y);
};

};
```

Figure 2.11: Many Chares: the interface file `hello.ci`

instance of the `Worker` chare. Instead of having to send it to them as an additional constructor parameter, we assign it to the global *read-only* variable `mainProxy` just once.

Charm++ does not allow you to use global variables, except when they are declared as `readonly`. Read-only variables are declared in the interface file by preceding their declaration by the keyword `readonly`. In addition, it is required that they can be set and changed only in construtors of main chares, or from functions/methods called directly (not asynchronously) from there. The RTS essentially takes a snapshot of the values of all read-only variables at the end of the main chare constructor, and copies them onto all physical processors (and expects you never to change themsubsequently, although it has no real way of checking this except in some debug modes).

Here, we initialize `mainProxy` in the `Master::Master` (line 18), and never change it anywhere else in the program. Each `Worker`, irrespective of which processor it is running on, has access to a valid copy of `mainProxy`, which it uses (line 38) to send results to the main chare. Note that `mainProxy` is not a keyword. It is just a variable that you could have named whatever you wanted.

The `Master` chare now creates ten `Worker` chare instances (line 16). What processor will they be created on? That is something that the Charm++ RTS will handle, and you shouldn't have to think about that. In later chapters, you will read about seed balancers and how the system manages to create these chares in a load-balanced manner. You will also learn of tehcniques to override the system placement of chares that allow you to specify where to create each chare. But our basic mode of operation is to let the system decide where to place chares.

```
#include <stdio.h>
#include "MyModule.decl.h"

CProxy_Master mainProxy; // readonly

/*mainchare*/
class Master: public CBase_Master
{
private:
    int count;
public:
  Master(CkArgMsg* m)
  {
      double pi = 3.1415;
      for (int i = 1; i <= 10; i++)
          CProxy_Worker::ckNew(i, pi);
      count = 10; // wait for 10 responses.
      mainProxy= thisProxy;
  };

    void printArea(int r, double area) {
        ckout << "Area of a circle of radius" << r << " is " << area << endl;
        count--;
        if (count == 0) CkExit();
    }

};

class Worker : public CBase_Worker
{
private:
    float y;
public:
    Worker( int r, double pi)
   {
       y = pi;
       ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
       mainProxy.printArea(r, y*r*r);
   }

};

#include "MyModule.def.h"
```

Figure 2.12: Many Chares: the C++ file `hello.C`

2.5.1 Example: Calculating Pi

Many of our programs have been using the value of π. For a slightly more realistic parallel program, we will now calculate the value of π using a master-slave approach.

The area of a circle of radius 1 unit is π, whereas the area of a square of size 2 units, within which such a circle is inscribed, is 4. So, if we throw random darts inside the square, and assuming the darts are thrown uniformly randomly within the square, the ratio of darts inside the circle to the total number of darts thrown should approximate $\pi/4$.

How do we throw darts in a computer? We will use simple random number generators that are provided by standard libraries included with C/C++. We will also throw darts in the positive quadrant, so that, the x and y coordinates of each dart placement is between 0 and 1.0.

A sequential program fragment for this purpose is shown below.

```
for (int i=0; i<numTrials; i++) {
  x = drand48();
  y = drand48();
  if ((x*x + y*y) < 1.0)
  inTheCircle++;
}
approxPi = 4.0 * (((double)inTheCircle)/((double)numTrials));
```

Suppose we decide that throwing a billion darts (`numTrials=1000000000`) will give us a good enough approximation. How many chares should we fire? If your inclination is to a billion chares, you are thinking like a good object-oriented sequential programmer, but not like a parallel programmer. Creating a chare takes some time, and messages may have to travel across processors for each. In fact, if we are creating chares from the main chare constructor (running on one of the processors), and you have seven processors in all, on average six out of seven chares will be created on a remote processor. Messages take microseconds to send and process on today's (2008) machines, but each trial is going to take only tens of nanoseconds at most. So, to be competitive with a sequential program, we must group together many dart-throwing trials in a single chare. How many? This is the issue of *grainsize control.* A rule of thumb is that the *average* work per message should be sufficiently high (say at least tens of microseconds) to amortize the overhead of messages. So, maybe 10,000 or 100,000 trials per chare will suffice. Notice that this consideration is *independent* of the total number of processors. With massive chare creation, we also have to consider the load balancing overhead. Let us choose to parameterize the computation, so we can experiment with the grainsize.

Figure 2.13 shows the interface file. The main chare will fire a bunch of `Worker` instances, who have only their constructor as their entry method. The main chare receives results from each `Worker` instance at its `addContribution` entry method. The result is simply two counts:

```
mainmodule MyModule {

  readonly CProxy_Master mainProxy;

  mainchare Master {
    entry Master(CkArgMsg *m);
    entry void addContribution(int numIn, int numTrials);
  };

  chare Worker {
    entry Worker(int numTrials);
  };

};
```

Figure 2.13: Calculating Pi: the interface file `hello.ci`

the number of trials conducted by that chare, and the number of darts that were inside the circle.

The first section of the `.C` file, with definition of the main chare, is shown in Figure 2.14. This is very similar to earlier programs (such as the earlier hello example), and so we leave it to you to read and understand it. The second section is shown in Figure 2.15 whichs shows the definition of the `Worker`. Each worker does its trials and sends the counts to the main chare.

You should now run this program, and watch if you end up calculating the value of π. Also, try providing different values on the command line for the `numChares` parameter, while holding the value of the `numTrials` fixed, *e.g.* at 100 million.

To measure the time the program takes, one can use the `CkWallTimer()` function provided by the Charm++. The `CkWallTimer()` function takes no parameters and returns a single `double` value which represents the number of seconds that has elapsed since the program began. This function can be used to measure the amount of time the calculate takes.

First, add a member variable of type `double` to the master class. Right before the `for` loop that creates the worker chares, store the value returned by a call to `CkWallTimer()` in the new member variable. This value represents the start time of the calculation. Second, right before the call to CkExit() in `Master::addContribution(...)`, insert a second call to the `CkWallTimer()` to get the end time of the calculation. By subtracting the start time from the end time, one can calculate the time taken by the calculation. It should be noted, however, that this will include the time taken to create the worker chare objects as well as the time to execute `Worker::Worker(...)`.

```
#include <stdio.h>
#include "MyModule.decl.h"

CProxy_Master mainProxy; // readonly

/*mainchare*/
class Master: public CBase_Master
{
private:
    int count, totalInsideCircle, totalNumTrials;
public:
  Master(CkArgMsg* m)
  {
      if (m->argc < 3) {
          ckout << "Need numTrials as a command line parameter" << endl;
          CkExit();
      }
      int numTrials = atoi(m->argv[1]);
      int numChares = atoi(m->argv[2]);
      if (numTrials % numChares) {
          ckout << "Need numTrials to be a divisible by numChares.. Sorry" << endl;
          CkExit();
      }
      for (int i = 0; i < numChares; i++)
          CProxy_Worker::ckNew(numTrials/numChares);
      count = numChares; // wait for count responses.
      mainProxy= thisProxy;
      totalInsideCircle = 0;
      totalNumTrials = 0;
  };

    void addContribution(int numIn, int numTrials) {
        totalInsideCircle += numIn;
        totalNumTrials += numTrials;
        count--;
        if (count == 0) {
            double myPi = 4.0* ((double) (totalInsideCircle))
                          / ((double) (totalNumTrials));
            ckout << "Approximated value of pi is:" << myPi << endl;
            CkExit();
        }
    }

};
```

Figure 2.14: Calculating Pi: `Main` chare in `hello.C`

```
class Worker : public CBase_Worker
{
public:
    Worker( int numTrials)
   {
       int inTheCircle = 0;
       double x, y;
       ckout << "Hello from a simple chare running on " << CkMyPe() << endl;

       for (int i=0; i< numTrials; i++)
       {
           x = drand48();
           y = drand48();
           if ((x*x + y*y) < 1.0)
               inTheCircle++;
       }
       mainProxy.addContribution(inTheCircle, numTrials);
   }

};

#include "MyModule.def.h"
```

Figure 2.15: Calculating Pi: `Worker` class in `hello.C`

Another point of consideration for this program is the generation of the random numbers. Each processor needs to generate a unique random seed. If two processors have the same initial seeds, they will generate the same sequence of random numbers. This can be accomplished by creating a function that sets the seed of the processors random number generator that is a function of the processor number. The function can be called once on each processor as the Charm++runtime system starts by marking it as an `initproc` function in the interface file. For more on `initproc` functions, please refer to the Charm++documentation.

2.5.2 Fibonacci: recursively creating chares

In the examples so far, only the main chare has been creating chares. We will now write a short program that illustrates reursive chare creation, and a typical divide-and-conquer program. We will use that traditional doubly-recursive algorithm for calculating n'th Fibonacci number, directly based on the following definition:

```
Fib(n) = if (n<2) n else Fib(n-1) + Fib(N-2)
```

This has an exponentioal complexity, and is a rather silly way of calculating n'th Fibonacci number, expecially since a program with *logarithmic* time complexity exists! But again, like many examples here, the purpose is to illustrate Charm++ rather than the parallel algorithm.

This example will also show how to use ".h" header files in Charm++ programs, and especially there relationship to the *decl* and *def* files that must be included at the right place. The example so far avoided header files because the classes and programs were quite simple, and we needed to show the example in as few lines of code as possible.

Figure 2.16 shows the interface file, while Figures 2.17 and **??** show the header (.h) file and the C++ file respectively. We include the `decl` file at the top of the header file, because the `decl` file declares some classes and types (such as `CProxy_fib`) that the header file uses. The `def` file is included at the bottom of the `.C` file.

If we were to fire chares for all values of $n > 1$, the chares will be doing very little work, and the overhead of creating and load-balancing them will be overlwhelming. To itroduce some terminology: "grainsize" (for our current purposes) is the amount of computation per parallel operation — which in this case, is creation of Fibonacci chares, and processing of each response from a child chare. Thus, our grainsize will be too small. So, instead we do some explicit "grainsize control": if n is below some pre-defined THRESHOLD, we will calculate Fib(n) using a sequential version of the (same, doubly recursive) algorithm.

ANother minor point to notice is the empty constructor

```
main(CkMigrateMessage *m) {}
```

The reaso for its existence has to do with migratable chares, which we will not encounter unitl a later chapter. For now, include this line as a necessary "incantation" in every chare definition.

```
mainmodule fib {

mainchare main
{
  entry main();
};

chare fib
{
  entry fib(int amIroot, int n, CProxy_fib parent);
  entry void response(int value);
};

};
```

Figure 2.16: Calculating N'th Fibonacci Number: the interface file `fib.ci`

```
#include "fib.decl.h"

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

class fib : public Chare
{
private:
  int num, result, count, IamRoot;
  CProxy_fib parent;
public:
  fib(CkMigrateMessage *m) {}
  fib(int amIRoot, int n, CProxy_fib parent);
  void response(int fibValue);
  void processResult();
};

```

Figure 2.17: Calculating N'th Fibonacci Number: the header file `fib.h`

```
#include "fib.h"
#define THRESHOLD 10

main::main(CkArgMsg * m)
{
  if(m->argc < 2) CmiAbort("./pgm N.");
  int n = atoi(m->argv[1]);
  CProxy_fib::ckNew(1, n, thishandle);
}

int seqFib(int n) {
  if (n<2) return n;
  else return (seqFib(n-1) + seqFib(n-2));
}

fib::fib(int AmIRoot, int n, CProxy_fib parent){
  IamRoot = AmIRoot;
  num = n;
  this->parent = parent;
  if (n< THRESHOLD) {
    result =seqFib(n);
    processResult();}
  else {
    CProxy_fib::ckNew(0,n-1, thishandle);
    CProxy_fib::ckNew(0,n-2, thishandle);
    result = 0;
    count = 2;  }
}

void fib::response(int fibValue) {
  result += fibValue;
  if (--count == 0) processResult();
}

void fib::processResult()
{
  if (IamRoot) {
    CkPrintf("fib(%d) = %d\n", num, result);
    CkExit();
  }
  else parent.response(result);
  delete this; /*this chare has no more work to do.*/
}

#include "fib.def.h"
```

Figure 2.18: Calculating N'th Fibonacci Number: the C++ file `fib.C`

Some Final Notes on the Code Examples in this Chapter

The code examples used throughout this chapter did not use header files to declare classes. Typically, classes are declared in a header file and the bodies of the member functions of that class are written in a separate code file. The examples in this chapter do not use a header file for the sake of brevity and having all the code in one place. The use of header files to declare classes is fully supported by Charm++. Refer to figure 2.1.

Chapter 3

Chares Arrays: Indexed Collections of Chares

In chapter 2, chare objects were introduced. Especially in large applications, create each of the individual chare objects one-by-one would be cumbersome to the programmer. Additionally, being able to collect a subset of the application's chare objects into *sets* or *collections* for the purpose of performing certain operations on all elements of the collection or using the collection's indexing scheme to more easily code patterns of communication between chare objects assists the programming in writing applications. In this chapter, we will introduce the first (and arguably the most widely used) type of collection in the Charm++ programming model, namely *chare arrays*.

As the name implies, a chare array is simply an array of chare objects. In some ways, chare arrays can be thought of as being similar to arrays of primatives in C/C++. For example, just as an individual integer in an integer array can be identified by using use the bracket operator ("[]") on the integer array, the paranthesis operator ("()") can be used on the chare array's proxy object to identify an individual chare object in the array of chare objects. Unlike primative arrays in C/C++, the individual elements of the chare array will be distributed across all of the available processing elements (PEs) available to the application. Further, chare arrays have other characteristics, such as their ability to be dense or sparse, that set them apart from primitive arrays in C/C++. Additionally, another difference is that chare arrays can be indexed using one dimension through six dimensions, using bit vectors, or using strings.

Because chare arrays are collections of active entities (i.e. chare objects) instead of simply being a collection of data storage locations (e.g. four byte integers), the programmers may also invoke an entry method on all of the members of the chare array, just as the programmer can invoke an entry method on a single chare object. The result is that every chare object within the array will (eventually) execute the entry method invoked by the programmer. This is commonly referred to as *broadcasting to the chare array*.

Before delving to deeply into the all of the characteristics and features of chare arrays, let us try to make the idea more concrete in the reader's mind via an example. We will begin with a single application that creates a single chare array, has each element in that array *do something*, and then exit once all of the chare array elements have completed their *task*.

3.1 Hello World from Array

In this section, we will present a simple example program that demonstrates the creation of a single chare array. In this example, as is the case with all Charm++ programs, starts with the runtime system creating an instance of the application's main chare object. The main chare object will then create a single chare array, the length of which can be specified using by the user on the program's command line. As the chare array is created, the constructors of each of the individual elements in the chare array (i.e. the chare objects) will be called. The constructor will print a simple message, "Hello World!" and invoke another entry method on the main chare object to indicate that the given chare object has been created. Once the last chare object *checks in* with the main chare object, the chare array will have completed it's initialization and the program will exit. Please note, it is not required that each element of a chare array check in with the main chare object. We do it here simply because that is our exit condition and thus we need to detect when the condition is met. Figure 3.1 gives an overview of the control flow of the overall program.

The example program is contained in two files: "simpleMain.ci" (the interface file) and "simpleMain.C" (the source file). Please note that we are leaving out a header file in this case simply because the program is short and we wanted to minimize the number of separate files. Figure 3.2 contains the contents of the interface file. The main chare class is called "Main" and is declared using the "mainchare" keyword to indicate that it is a main chare class instead of just a standard chare care. Remember, a main chare is the same as a standard chare except for one difference, an instance of it will be created automatically by the runtime system when the application is started. The chare array class is called "Hello" (that is, the each element in the chare array will be an instance of the "Hello" chare class). The "[1D]" indicates that the array will be indexed as a single dimensional array where the indexes are integers. Besides being declared as a 1D array, note that the declaration of the "Hello" chare array class is declared in much the same way that a singleton chare class is declared.

Figures 3.3 and 3.4 contain code from the "simpleMain.C" source code file for the "Main" and "Hello" classes, respectively. For the sake of brevity, some portions of the code will be left out.

When the program begins execution, the constructor of the main chare object is invoked by the runtime system (see Figure 3.3). The first thing the constructor does is read the command line arguments given to the program. In this case, the program will accept a single

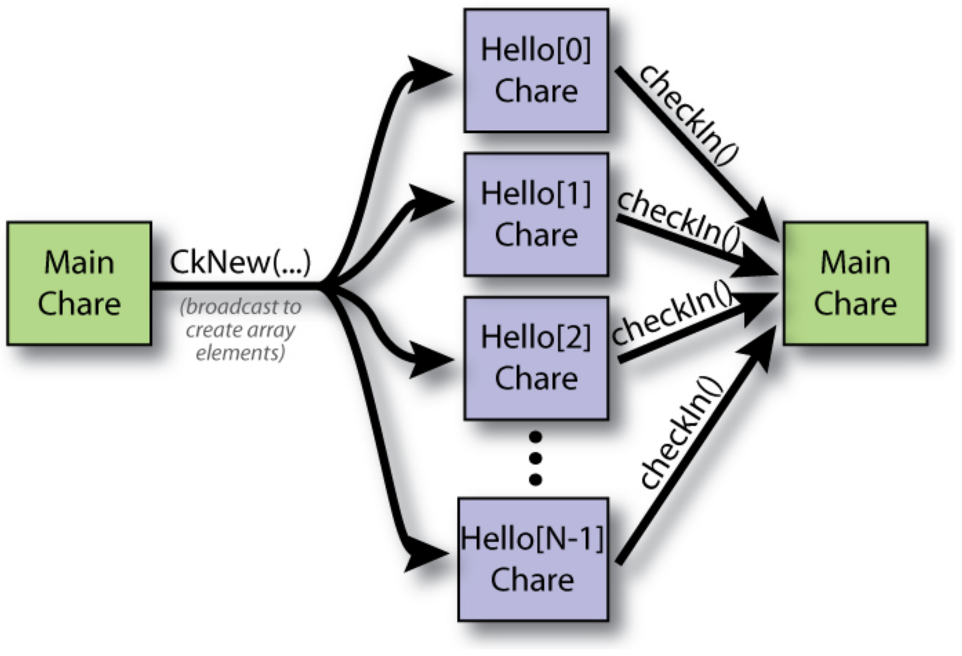

Figure 3.1: Control flow of array based "Hello World!" example program.

```
mainmodule simpleMain {

  mainchare Main {
    entry Main();
    entry void checkIn();
  };

  array [1D] Hello {
    entry Hello(CProxy_Main mainProxy);
  };
}
```

Figure 3.2: Simple "Hello World" Chare Array Program: the C file "simpleMain.ci"

```
class Main : public CBase_Main {

 private:
  int helloArraySize;

 public:
  Main(CkMigrateMessage *msg) { }
  Main(CkArgMsg* msg) {

    // Check to see if there are any command line arguements
    helloArraySize = DEFAULT_HELLO_ARRAY_SIZE;
    if (msg->argc > 1) helloArraySize = atoi(msg->argv[1]);
    if (helloArraySize <= 0) {
      CkPrintf("[ERROR] :: There should be at least one "
               "element in the Hello array... exiting.\n");
      CkExit();
    }

    // No longer need the message object, so delete it
    delete msg;

    // Create the chare array with arraySize elements
    CProxy_Hello helloArray =
      CProxy_Hello::ckNew(thisProxy, helloArraySize);
  }

  void checkIn() {
    static int checkInCount = 0;
    if ((++checkInCount) >= helloArraySize) CkExit();
  }
};
```

Figure 3.3: Simple “Hello World” Chare Array Program: Main chare class in the C file “simpleMain.C”

command line argument indicating the length of the chare array it will create. If no value is given on the command line, it will simply create a chare array of some default size. Next, the constructor deletes the message it received since it will no longer need any more information from the message. Finally, the constructor creates an instance of the chare array by calling the static "CkNew()" function on the chare array's proxy object. The last argument to the CkNew() function ("helloArraySize") indicates the number of elements that the chare array will contain. The parameters before this value (just one in this case, "thisProxy") are passed as parameters to the constructors of each of the array elements as they are created on the target processors. Here, the main chare object's proxy object will be passed as a parameter to each off the array elements' constructors. Remember, a chare object's proxy object allows any other object to interact with the chare object, even if they are located on different physical processors.

```
class Hello : public CBase_Hello {

 public:
  Hello(CkMigrateMessage *msg) { }
  Hello(CProxy_Main mainProxy) {
    CkPrintf("Hello World! [%d](%d)\n", thisIndex, CkMyPe());
    mainProxy.checkIn();
  }
};
```

Figure 3.4: Simple "Hello World" Chare Array Program: Hello chare class in the C file "simpleMain.C"

Figure 3.4 contains the code for the Hello chare class. This class is quite simple. The constructor simply displays the text "Hello World!" to the user by calling the CkPrintf() function. Then, it invokes the "Main::checkIn()" entry method on the main chare object, indicating that the given array element has been created. The constructor takes a single arguement, a proxy object for the main chare object. This is required because the chare array elements needs a copy of the proxy object in order to invoke entry methods on the main chare object.

Finally, figure 3.3 contains the code for the "checkIn()" entry method. Each time this entry method is invoked, the counter "checkInCount" is incremented. When this counter reaches the length of the chare array, "helloArraySize" set in the main chare object's constructor, all of the elements in the chare array will have completed their constructors and the main chare will call "CkExit()" to cause the program to terminate.

Figure 3.5 contains some example output generated when the simple "Hello World!"

program is executed. In this example, the "charmrun" command is used to launch the Charm++ program. The "+p2" command line arguement indicates that two processors are to be used to execute the program. The "./simpleHello" command line arguement is the executable to run. The remaining command line arguements, in this case "10", are passed to the Charm++ program and processed by it (through the CkArgMsg parameter in the main chare's constructor, see "Main::Main()" in figure 3.3). Each "hello World!" line also contains two numbers. The first number, enclosed in square brakets ("[]") indicates the index of the chare array element that generated the given line of output (i.e. the value of "thisIndex"). Each chare in a chare array has a unique index and the value of that index is contained in the variable "thisIndex." The second number, enclosed in parenthesis ("()"), indicates the physical processor that executed the constructor.

```
$ ./charmrun +p2 ./simpleHello 10
Hello World! [0](0)
Hello World! [1](1)
Hello World! [2](0)
Hello World! [3](1)
Hello World! [4](0)
Hello World! [5](1)
Hello World! [6](0)
Hello World! [7](1)
Hello World! [8](0)
Hello World! [9](1)
```

Figure 3.5: Example output of Simple "Hello World" Chare Array Program

At the time of chare array creation, the Charm++runtime system assigns the elements of the chare array to the processors using a round-robin mapping. Round-robin mapping is not the only mapping scheme available in Charm++, but is the default scheme used for any chare array where the programmer has not specified the mapping scheme explicitly. In round-robin, the first element is mapped to the first processor, the second element is mapped to the second processor, and so on. If there are more chare array elements than processors, then when the final processor is reached, the runtime system begins with first processors once again and continues through until the last, cycling through the list of processors as many times as is required so that all of the chare array elements have been assigned to a physical processor.

[DMK : TODO : Add a figure to illustrate round-robin mapping ???]

At this point, we encourage the reader to write the code for the simple hello array program and try executing it, using the provided code listings as a guide. It should be noted that the output lines may appaer in a different order each time the program is executed, especially as more and more cores are used. To understand why this is, first consider that there is no particular order being imposed on the creation of the chare array elements. It is not necessary for the first element to be created before the second, the second before the third, and so on. Because there is no enforced order in which the constructors are being called, there is not enforced ordering to the various calls made to "CkPrintf()." Second, as an optimization to I/O performance, the output generated by the various processors may be buffered by the runtime system, causing the ordering of "CkPrint()" output to appear different to the user than the order of the actual calls themselves. For example, processor X prints message A and that message is buffered. Then, processor Y prints message B and then flushes its I/O buffer, causing message B to be displayed to the user. Finally, processor X prints message C and flushes its I/O buffer, causing messages A and C to be displayed to the user. Even though the application printed the messages in the order A, B, and then C, the user will see the output as B, A, and then C because of the interference of the I/O buffering mechanism. To programmers, especially those that are new to parallel programming in general, the reordering of output can be quite tricky, causing them to think that their program executed differently than it actually did. We will see how including certain information in the output can be useful to understanding the output, even if the lines of output generated by the program occur in a different order each time the program is executed.

3.2 A Single Ring

The next example, "Single Ring," illustrates how to create and use chare arrays. In this program, a single chare array, "the ring," will be created. Consecutive array elements are considered connected along with the first and last array elements, thus creating the "ring."

The program works as follows (refer to figure 3.6). The main chare will instruct one of the elements, at random, to begin a ring traversal by invoking the "doSomething()" entry method on the chosen element (*(1: begin)* in figure 3.6). At the same time, the total number of ring traversals that should be completed will also be indicated. The "doSomething()" entry method will print a message, indicating that the entry method was called and giving some information about the specific invocation. Once the message has been printed, it will instruct the next element in the array to do the same thing by invoking "doSomething()" on it also (the last element will invoke "doSomething()" on the first element, creating the ring). A traversal of the ring is complete after each element in the array has executed "doSomething()." This process will continue until the specified number of traversals have completed (*(2: begin next traversal)* in figure 3.6). Once the all of the traversals have been completed, an entry method on the main chare will be invoked causing the program to exit (*(3: end)* in figure 3.6). At this point, we could just as easily call "CkExit()" from within the body of "doSomething()," and so the decision to invoke an entry method on the main chare object may seem a bit arbitrary

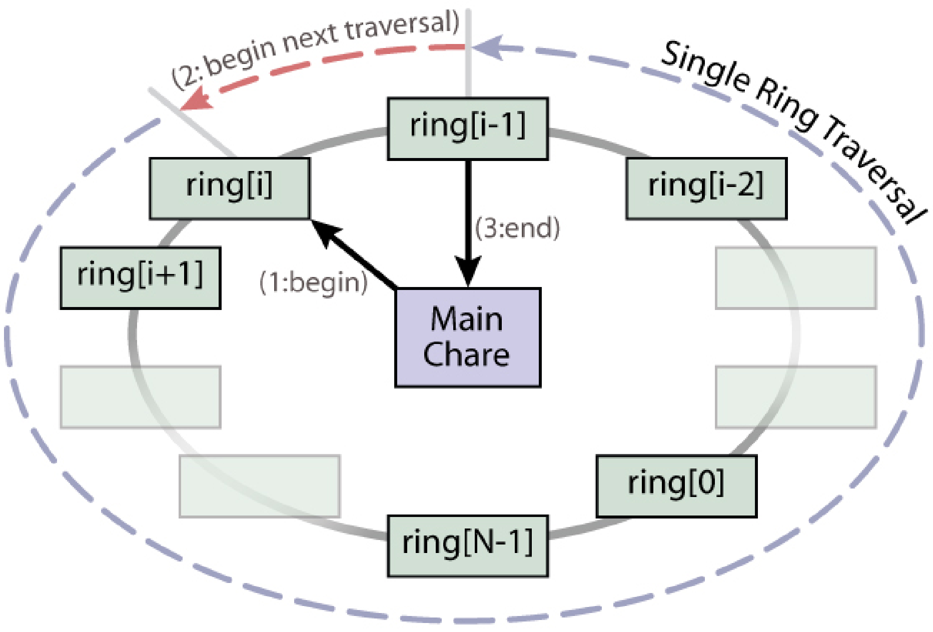

Figure 3.6: Control flow of Single Ring example program. In this figure, there are N array elements in the ring where index i is the element that is chosen by the main chare to begin the ring traversals.

(and unecessary at this point). However, we will expand on this example in the future, and this decission should be more clear at that time.

Figure 3.7 contains the code for the interface (.ci) file for the single ring program. It is similar to the previous interface file (see figure 3.2. The main difference for this example is that the chare array class has both a constructor and an entry method. Entry methods for chare arrays are declared exactly the same way they are declared for single chare objects.

```
mainmodule arrayRing {

  mainchare Main {
    entry Main();
    entry void ringFinished();
  };

  array [1D] Ring {
    entry Ring(CProxy_Main mp, int rs);
    entry void doSomething(int elementsLeft, int tripsLeft,
                           int fromIndex, int fromPE);
  };

}
```

Figure 3.7: “Single Ring” Chare Array Program: Interface file “arrayRing.ci”

Figure 3.8 contains the code for the main chare object in the single ring program. The code here is fairly similar to the array based “Hello World!” program presented in section 3.1. The main differences are that the main chare object’s constructor initiates the work in the chare array by sending one of the array elements a message and the way in which the program terminates. Only one element will invoke an entry method on the main constructor, instead of all of the elements.

As can be seen at the end of the Main::Main() constructor in figure 3.8, the main chare creates an instance of the “Ring” chare array (lines 34-35). This causes the Charm++ runtime system to begin creating the array elements on the remote processors and calling their constructors, using the parameters passed into the CProxy_Main::CkNew() call just as it was described in section 3.1. The main chare’s constructor also sends a message to one of the array elements, at random, to initiate the ring traversals (lines 37-38). The Charm++programming model does not guarantee that messages are received in same order in which they are sent. However, it is ensured that the constructor of any given chare will be executed before any entry methods are invoked on that chare object. Thus, the programmer does not need to

```
class Main : public CBase_Main {
 public:

  Main(CkMigrateMessage *msg) { }
  Main(CkArgMsg* msg) {

    // Check to see if there are any command line arguements
    int ringSize = DEFAULT_RING_SIZE;
    int tripCount = DEFAULT_TRIP_COUNT;
    if (msg->argc > 1) ringSize = atoi(msg->argv[1]);
    if (msg->argc > 2) tripCount = atoi(msg->argv[2]);
    if (ringSize <= 0 || tripCount <= 0) {
      CkPrintf("[ERROR] :: Invalid ringSize or tripCount\n");
      CkExit();
    }
    delete msg;  // Done with message (so delete it)

    // Display some information to the user about this run
    CkPrintf("\"Array Ring (Single)\" Program\n");
    CkPrintf("  ringSize = %d, tripCount = %d, #PEs() = %d\n",
             ringSize, tripCount, CkNumPes());

    // Create the chare array with arraySize elements and
    //  tell a random element to start doing something
    CProxy_Ring ring =
      CProxy_Ring::ckNew(thisProxy, ringSize, ringSize);
    srand(time(NULL));  // Initialize random number generator
    ring(rand() % ringSize)
      .doSomething(ringSize, tripCount, -1, -1);
  }

  void ringFinished() { CkExit(); }
};
```

Figure 3.8: “Single Ring” Chare Array Program: Main chare class in the C file “arrayRing.C”

perform any special synchronization in the main chare's constructor between the creation of the chare array (call to "CkNew") and invoking an entry method on one of the elements (call to "doSomething()").

Figure 3.9 contains the code for the "Ring" chare array class. The constructor is fairly straight forward. It simply stores the parameters it's given into member variables for later use. The nextI() member function is also fairly straight forward. It returns the array index of the next array element in the ring.

The "Ring::doSomething()" function is a bit more involved. It starts by displaying some text to the user. The text indicates which object in the ring is currently "doing work," which ring traversal this message belongs to (related to "tripsLeft"), and which array element told this element to "do something." The entry method then has to determine what to do next. If there are elements left in this traversal of the ring (i.e. "elementsLeft >= 2"), then it simply tells the next array element to "do something" and decrements the "elementsLeft" counter. Otherwise, if there are no elements left in this traversal and there are more traversals remaining, start the next traversal by reseting the "elementsLeft" counter and decrementing the "tripsLeft" counter. Finally, if there are no elements left in this traversal and no more traversals remaining, invoke the "ringFinished()" entry method on the the main chare object indicating that the ring has completed all of it's traversals. *Please note, both "elementsLeft" and "tripsLeft" are 1 based and include the current call (e.g. "elementsLeft == 1" means this ivocation is the one remaining call in the traversal).* When the "Main::ringFinished()" entry method is called, "CkExit()" is called causing the program to terminiate.

Figure 3.10 contains an example execution of the "Simple Ring" program. Once again, the "charmrun" command is used to launch the Charm++ program on multiple processors. The "+p2" indicates that two processors should be used to run the program. "./arrayRing" is the name of the Charm++ executable. The remaining command line parameters are passed to the Charm++ program (in this case "5" and "3", see "Main::Main()" in 3.8). The "5" indicates that there should be five array elements in the ring. The "3" indicates that there should be three full ring traversals before the program exits.

The output of the program in figure 3.10 should be read as follows. The values in the brackets ("[]") indicate the array index of the array element printing that particular output line. The values in parenthesis ("()") indicate the physical processor on which the array element is located. The "tripsLeft" value indicates how many full ring traversals (including the current traversal) still remain before the program exits. The "from" portion of the line indicates which array element sent the message to cause *this* array element to print *this* line. For the element that is invoked first, starting the initial traversal, the values in the "from" portion of the line are "[-1](-1)" indicating that the main chare object actually sent the message to start the first traversal. See the code figures to understand the details on what the code is doing exactly.

```
class Ring : public CBase_Ring {
 private:
  CProxy_Main mainProxy;  // Proxy object for the main chare
  int ringSize;           // Number of elements in the ring

 public:
  Ring(CkMigrateMessage *msg) { }
  Ring(CProxy_Main mp, int rs) {
    mainProxy = mp;
    ringSize = rs;
  }

  inline int nextI() { return ((thisIndex + 1) % ringSize); }

  void doSomething(int elementsLeft, int tripsLeft,
                   int fromIndex, int fromPE) {

    // Do something (display some text for the user)
    printf("Ring[%d](%d): tripsLeft = %d, from [%d](%d)\n",
           thisIndex, CkMyPe(), tripsLeft, fromIndex, fromPE);

    // Send message to continue traversals or notify main
    if (elementsLeft > 1) {  // elements left in traversal
      thisProxy(nextI()).doSomething(elementsLeft - 1,
        tripsLeft, thisIndex, CkMyPe());
    } else if (tripsLeft > 1) {  // start next traversal
      thisProxy(nextI()).doSomething(ringSize,
        tripsLeft - 1, thisIndex, CkMyPe());
    } else { // otherwise, all traversals finished
      mainProxy.ringFinished();
    }
  }
};
```

Figure 3.9: “Single Ring” Chare Array Program: Ring chare class in the C file “arrayRing.C”

```
$ ./charmrun +p2 ./arrayRing 5 3
"Array Ring (Single)" Program
  ringSize = 5, tripCount = 3, #PEs() = 2
Ring[0](0): tripsLeft = 3, from [-1](-1)
Ring[1](1): tripsLeft = 3, from [0](0)
Ring[2](0): tripsLeft = 3, from [1](1)
Ring[3](1): tripsLeft = 3, from [2](0)
Ring[4](0): tripsLeft = 3, from [3](1)
Ring[0](0): tripsLeft = 2, from [4](0)
Ring[1](1): tripsLeft = 2, from [0](0)
Ring[2](0): tripsLeft = 2, from [1](1)
Ring[3](1): tripsLeft = 2, from [2](0)
Ring[4](0): tripsLeft = 2, from [3](1)
Ring[0](0): tripsLeft = 1, from [4](0)
Ring[1](1): tripsLeft = 1, from [0](0)
Ring[2](0): tripsLeft = 1, from [1](1)
Ring[3](1): tripsLeft = 1, from [2](0)
Ring[4](0): tripsLeft = 1, from [3](1)
```

Figure 3.10: Example output of “Single Ring” Chare Array Program

3.3 Multiple Rings

In this next example, we will be exploring a slightly more complicated version of the "Single Ring" example program from section 3.2. This example will show how a program can have multiple sets of chare objects that are performing parallel computations that are largely independent of one another. To keep the example simple we will simply duplicate the *ring* from the "Single Ring" example. However, there is nothing stoping the programming from having completely different computations going on in each set of chare objects.

The major differences between this "Multiple Ring" and the previous "Single Ring" examples are as follows. First, this example has multiple rings, each represented by a chare array just as before. Second, as messages are moving around each ring, they will skip a random number of array elements instead of simplying moving from one element to the next consecutive element. As a result, even though each ring will be performing the same computation (executing the same code), the example will show, in a very simple sense, that each computation is independent from the others. Third, to demonstrate modularity, the modules (interface) files and source code files for the chare classes will be divieded into multiple files.

Figure 3.11 illustrates the program flow of the "Multiple Ring" example program. It is very similar to the "Single Ring" program. The basic difference in the control flow is that the main chare will create several rings, instead of just one. Since there are multiple rings, the program will only exit after all of the rings have completed all of their traversals.

Figure 3.12 contains the interface file for the "Main" chare class. This interface file is fairly similar to previous interface files that have been presented so far with one notable exception, the "extern" line. The extern line indicates that there is yet another module (in another interface file) called "multiRing_Ring" that should be *included* by this module. The contents of the "multiRing_Ring" module will be declared in a different interface file (see figure 3.13). Both chare classes could have easily been declared in a single interface file, we have done this simply to provide an example of multiple interface files, since modularity is useful as applications grow more complex and/or when pieces of an application are developled independently of one another.

Figure 3.14 contains some of the source code for the Main chare class. Some of the code, parsing the command line, displaying usage information to the user, and so on has been left out for brevity. The user can indicate on the command line how many rings (chare arrays) should be created (*numRings*), the number of elements in each ring (*ringSize[i]*), and the number of traversals around each ring that should be completed (*tripCount[i]*). Once the command line has been processed, a header is printed for the user displaying some information about the details of the program execution. Finally, each ring is created using "CkNew()" and the first traversal of each ring is started by calling the "doSomething()" entry method on a random element of the ring.

Figure 3.15 contains the code for the Ring chare class. Once again, this code is basically the same as it was in the "Single Ring" example (see section 3.2). The main difference is the

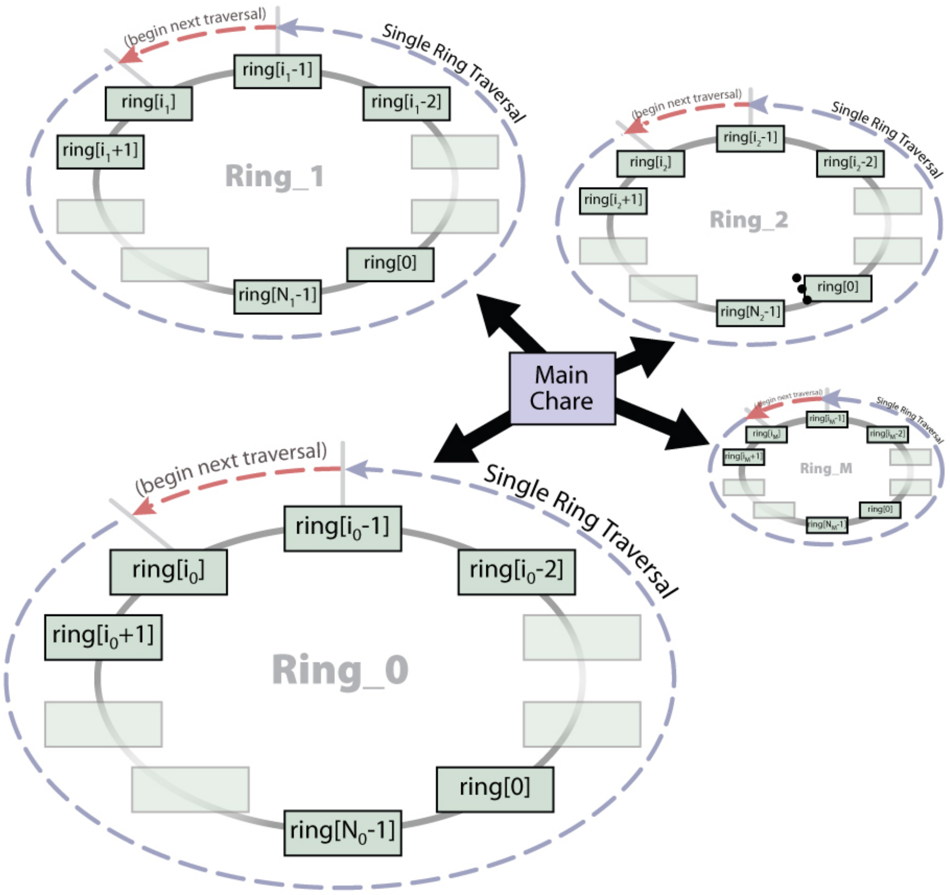

Figure 3.11: Control flow of Multiple Ring example program. Refer to figure 3.6 in section 3.2.

```
mainmodule multiRing_Main {

  // Declare chare objects/collections for the Main module
  mainchare Main {
    entry Main();
    entry void ringFinished();
  };

  // Include the Ring object's module
  extern module multiRing_Ring;

}
```

Figure 3.12: "Multi-Ring" Chare Array Program: Main chare class interface file "multiRing_Main.ci"

```
module multiRing_Ring {

  // Include the Main object's module (main proxy reference)
  extern module multiRing_Main;

  // Declare chare objects/collections for the Ring module
  array [1D] Ring {
    entry Ring(CProxy_Main mp, int rs, int rID);
    entry void doSomething(int elementsLeft, int tripsLeft,
                           int fromIndex, int fromPE);
  };

}
```

Figure 3.13: "Multi-Ring" Chare Array Program: Ring chare class interface file "multiRing_Ring.ci"

```
class Main : public CBase_Main {
 private:
  int numRings;

 public:
  Main(CkMigrateMessage *msg) { }
  Main(CkArgMsg* msg) {
    int *ringSize = NULL, *tripCount = NULL;
    processCommandLine(msg, &ringSize, &tripCount);

    CkPrintf("\"Array Ring (Multi)\" Program\n");
    CkPrintf("  numRings = %d, #Pes() = %d\n",
             numRings, CkNumPes());
    for (int i = 0; i < numRings; i++)
      CkPrintf("  Ring_%d : ringSize = %d, tripCount = %d\n",
               i, ringSize[i], tripCount[i]);

    // Create the rings
    for (int i = 0; i < numRings; i++) {
      CProxy_Ring ring = CProxy_Ring::ckNew(
        thisProxy, ringSize[i], i, ringSize[i]);
      ring(rand() % ringSize[i]).doSomething(
        ringSize[i], tripCount[i], -1, -1);
    }
  }

  void processCommandLine(CkArgMsg* msg, int** rs, int** tc);
  void printUsage(const char* const errStr, char* appName);

  void ringFinished() {
    static int finishedCount = 0;
    if ((++finishedCount) >= numRings) { CkExit(); }
  }
};

```

Figure 3.14: “Multi-Ring” Chare Array Program: Main chare class in the C file “multiRing_Main.C”

introduction of the *skipAmount* variable. In the “Single Ring” example, when an element received a message, it simply sent a message to the next consecutive element in the ring. However, in this example, upon receiving a message, the array element will send a message to another random element in the ring furthur along in the ring traversal. That is, if half of the ring has been traversed already, the current array element will send the next message to one of the element in the other half of the array at random.

Another strength of the Charm++program model is that various portions of the application can be decomposed independently of one another. That is to say, the chare objects in the application are spread across the individual processors, not tied to specific processors. If, for example, an application were to use a parallel library, the application developers do not have to concern themselves with how the library works internally or which processors the library will use. Instead, they simply write their application specific code, decomposing it as they see fit, and make calls in to the parallel library. The mapping of the objects to processes (and load balancing) is left up to the runtime system freeing the programmers of this responsibility. In this particular example, “Multiple Rings,” the main chare object can instantiate as many rings of any size that it wishes to instantiate without worrying about how they interact with one another. The runtime system is free to migrate these chare array elements between processors to balance the overall load and thus optimize the overall program performance.

Figure 3.16 contains the output from a single execution of this example program. One may find the ordering of the output to be quite confusing. We have already addressed why lines in the output of Charm++applications may be in a different order than the actual exection of the code itself for the reasons discussed at the end of section 3.1. As the programs get more complex and have more and as more *asynchronous events* are going on, the ordering of the output is more likely to get mixed up compared to the order in which the print calls are actually made.

Figure 3.17 contains the exact same output, though the ordering of the lines has been rearranged in to an order that one might expect. This is where the additional “from” information on each line is useful. The output of each ring (“Ring_0,” “Ring_1,” and so on) is grouped together. In this example, the rings do not communicate with one another so there is no particular ordering or dependency between the output lines for different rings. Within each ring’s output, each line indicates which element is outputting the line (line starts with “Ring_$\alpha[\beta](\delta)$” where α identifies which ring the output line came from, γ identifies which array element in the ring displayed the line, and δ indicates which processor the array element is located on). The “from” portion of the line follows the same convention (with the same ring being assumed). The addition of this “from” information to each lines helps the user understand what is going on in the execution a bit easier than if it wasn’t there (how the messages were actually passed between elements of the various rings). The “tripsLeft” counter indicates which traversal of the ring this is (i.e. once the message makes it all the way around the ring, the “tripsLeft” counter is decremented).

```
class Ring : public CBase_Ring {
 private:
  CProxy_Main mainProxy;  // Proxy object for the main chare
  int ringSize;           // Number of elements in the ring
  int ringID;             // ID value for this ring

 public:
  Ring(CkMigrateMessage *msg) { }
  Ring(CProxy_Main mp, int rs, int rID)
    : mainProxy(mp), ringSize(rs), ringID(rID) { }

  int nextI(int s) { return ((thisIndex+s) % ringSize); }

  void doSomething(int elementsLeft, int tripsLeft,
                   int fromIndex, int fromPE) {

    // Do something (display some text for the user)
    printf("Ring_%d[%d](%d): tripsLeft = %d, from [%d](%d)\n",
           ringID, thisIndex, CkMyPe(), tripsLeft, fromIndex,
           fromPE);

    // Send message to continue traversals or notify main
    if (elementsLeft > 1) {  // elements left in traversal
      int skipAmount = (rand() % elementsLeft) + 1;
      thisProxy(nextI(skipAmount)).doSomething(
        elementsLeft - skipAmount, tripsLeft, thisIndex,
        CkMyPe());
    } else if (tripsLeft > 1) {
      thisProxy(nextI(1)).doSomething(
        ringSize, tripsLeft - 1, thisIndex, CkMyPe());
    } else {
      mainProxy.ringFinished();
} } };
```

Figure 3.15: “Multi-Ring” Chare Array Program: Ring chare class in the C file “multiRing_Ring.C”

```
$ ./charmrun +p3 ./multiRing 3 5 3 10 1 8 2
Ring_0[1](1): tripsLeft = 3, from [-1](-1)
Ring_2[7](1): tripsLeft = 2, from [-1](-1)
Ring_1[7](1): tripsLeft = 1, from [2](2)
Ring_0[1](1): tripsLeft = 2, from [0](0)
Ring_0[4](1): tripsLeft = 2, from [1](1)
Ring_0[1](1): tripsLeft = 2, from [4](1)
Ring_2[7](1): tripsLeft = 1, from [6](0)
Ring_2[1](1): tripsLeft = 1, from [7](1)
Ring_2[4](1): tripsLeft = 1, from [3](0)
"Array Ring (Multi)" Program
  numRings = 3, #Pes() = 3
  Ring_0 : ringSize = 5, tripCount = 3
  Ring_1 : ringSize = 10, tripCount = 1
  Ring_2 : ringSize = 8, tripCount = 2
Ring_0[0](0): tripsLeft = 3, from [1](1)
Ring_2[6](0): tripsLeft = 2, from [7](1)
Ring_0[0](0): tripsLeft = 1, from [2](2)
Ring_2[3](0): tripsLeft = 1, from [1](1)
Ring_2[6](0): tripsLeft = 1, from [4](1)
Ring_1[8](2): tripsLeft = 1, from [-1](-1)
Ring_1[2](2): tripsLeft = 1, from [8](2)
Ring_0[2](2): tripsLeft = 1, from [1](1)
Ring_0[2](2): tripsLeft = 1, from [0](0)
```

Figure 3.16: Example output of “Multi-Ring” Chare Array Program

```
$ ./charmrun +p3 ./multiRing 3 5 3 10 1 8 2
"Array Ring (Multi)" Program
  numRings = 3, #Pes() = 3
  Ring_0 : ringSize = 5, tripCount = 3
  Ring_1 : ringSize = 10, tripCount = 1
  Ring_2 : ringSize = 8, tripCount = 2

Ring_0[1](1): tripsLeft = 3, from [-1](-1)
Ring_0[0](0): tripsLeft = 3, from [1](1)
Ring_0[1](1): tripsLeft = 2, from [0](0)
Ring_0[4](1): tripsLeft = 2, from [1](1)
Ring_0[1](1): tripsLeft = 2, from [4](1)
Ring_0[2](2): tripsLeft = 1, from [1](1)
Ring_0[0](0): tripsLeft = 1, from [2](2)
Ring_0[2](2): tripsLeft = 1, from [0](0)

Ring_1[8](2): tripsLeft = 1, from [-1](-1)
Ring_1[2](2): tripsLeft = 1, from [8](2)
Ring_1[7](1): tripsLeft = 1, from [2](2)

Ring_2[7](1): tripsLeft = 2, from [-1](-1)
Ring_2[6](0): tripsLeft = 2, from [7](1)
Ring_2[7](1): tripsLeft = 1, from [6](0)
Ring_2[1](1): tripsLeft = 1, from [7](1)
Ring_2[3](0): tripsLeft = 1, from [1](1)
Ring_2[4](1): tripsLeft = 1, from [3](0)
Ring_2[6](0): tripsLeft = 1, from [4](1)
```

Figure 3.17: Modified example output of “Multi-Ring” Chare Array Program

3.4 Reductions

This section will cover the basics of performing simple reductions on an array of chare objects in the Charm++ programming model. Before we discuss how to perform a reduction, we first start by describing what a reduction is.

As the name implies, a reduction is an operation that reduces a large set of values into a smaller set of values (typically, many values reduced to only a single value) according to a specified operation. For example, consider an array of integers and imagine that we want to calculate the sum of the all integers in that array. We would repeatedly apply the operation to the values within the array, along with any intermediate values that we have created during the calculation, until we processed all of input and arrived at a single final value. One of the other features of a reduction is that the operation being applied has to be both communative[1] and associative[2]. This allows the operation to be applied to the individual values within the original set in any order.

The function *reduceIntArray_Addition_1* in figure 3.18 is an example of the typical way in which the sum of an array of integers is calculated in a serial program (i.e. a reduction applying the 'add' operation is typically performed). The function body is straight forward. An intermediate value, r, is created and initialized to the identity value for addition.[3] The loop then iterates over each value in the array, adding the value of the array element to the intermediate value. Once all values have been added to r, the value of r is returned as the sum of all the values in the input array a.

Because the operation being used in this reduction ('add') is both communative and associative, we can modify the code of *reduceIntArray_Addition_1* to perform the operations on the values in a different order. In particular, we can divide the array into two equal halves and calculate the sum of each of those halves (refer to *reduceIntArray_Addition_2* in figure 3.18). In a sequential program, doing this doesn't make much sense, but in the context of creating parallel applications this begins to make sense. The *for-i* and *for-j* loops are data independent of one another and thus can be performed in parallel. The results of the two different loops still need to performed in order to get the final result, using the same operation applied to the initial values themselves. In *reduceIntArray_Addition_2*, we have only divided the work into two equal halves, however, the reduction computation could be broken down into even smaller parts.

Chare arrays have a natural associate with reductions. Chare arrays are a set of chare

[1]The operation $\otimes$ is said to be communative if $A \otimes B = B \otimes A$. Multiplication is an example of a communative operation (e.g. $2 * 3 = 3 * 2$). Division is an example of an operation which is not communative (e.g. $2 \div 1 \neq 1 \div 2$).

[2]The operation $\oplus$ is said to be associative if $(A \otimes B) \otimes C = A \otimes (B \otimes C)$. Addition is an example of an assocative operation (e.g. $2 + 3 = 3 + 2$). Subtraction is an example of an operation which is not associative (e.g. $(5 - 3) - 2 \neq 5 - (3 - 2)$).

[3]An identity value, I, of an operation, $\otimes$, is a value such that $a \otimes I = a$. For example, zero is the identity value of addition ($a + 0 = a$), and one is the identity value of multiplication ($a * 1 = a$).

```
// Perform a reduction on an interger array using the 'add' operation
//   r = a[0] + a[1] + a[2] + a[3] + a[4] + ... + a[aLen-1]
int reduceIntArray_Addition_1(int *a, int aLen) {
  int r = 0;
  for (int i = 0; i < aLen; i++) {
    r = r + a[i];
  }
  return r;
}

// Perform a reduction on an integer array using the 'add' operation
//   r = (a[0] + ... + a[aLen/2]) + (a[aLen/2+1] + ... a[aLen-1])
int reduceIntArray_Addition_2(int *a, int aLen) {

  // For-i loop : Add the values in the first half of intArray
  int r_A = 0;
  for (int i = 0; i < aLen / 2; i++) {
    r_A = r_A + a[i];
  }

  // For-j loop : Add the values in the second half of intArray
  int r_B = 0;
  for (int j = (aLen / 2) + 1; j < aLen; j++) {
    r_B = r_B + a[i];
  }

  // For the final result, add the results of for-i and for-j
  return = r_A + r_B;
}
```

Figure 3.18: Two versions of a function that performs a serial reduction on an array of integers, applying the 'add' operation. [*DMK - Should add a figure showing 'reduction trees' graphically?*]

objects where, at least in some cases, it is useful to calculate some statistic (e.g. an average), a maximum error in data distributed across the chares (e.g. applying a maximum-like function), and so on. In section 3.5, we will see one such application, a 5-point stencil applications, which uses a reduction to calculate the maximum value change between timesteps in order to test for convergence and thus completion. First, let us look at how reductions can be performed on chare arrays.

3.4.1 Basic Reduction

This simple example will illustrate how to perform a reduction over a chare array in Charm++. First, the main chare object will create a chare array as we have already demonstrated. Once created, all of the elements in the chare array will contribute an integer value (in this case, their own index in the chare array). The operation that will be applied is addition (called *sum_int* in the code). Finally, when the reduction has completed, the main chare object will be notified of the result via a *callback*, which it will display just before exiting the program.

Figure 3.19 contains both the interface and header files for the main chare object. This is a fairly straight forward main chare object similar to the previous main chare objects that have been presented in previous examples. Figure 3.20 contains the source code file. Similar to previous example's main chare objects, the main chare's constructor starts by processing the command line, displaying some information for the user, and creating a chare array. The difference with this example is that once the chare array has been created, but before any entry methods are invoked on any of it's elements (i.e. the broadcast to "doWork()"), the main chare object registers one of it's entry methods as a reduction client for the array. This is done using "CkCallback." A callback object identifies an action to take (call an entry method, call a member function, broadcast to a chare array, do nothing, etc.) once the runtime has completed a specified task for the application. In this case, the main chare object's "reductionCallback" entry method is specified as the target for the callback, and the callback is registered with the array as the callback to activate whenever a reduction on the "elems" chare array completes. In other words, once all of the elements of the "elems" chare array have called "contribute," the runtime system will apply the specified reduction function (e.g. "sum_int") and pass the value back to the program via the target of the callback object (e.g. the main chare object's "reductionCallback" entry method). Since this example is only meant to illustrate how one goes about using a single reduction in a Charm++application, the "reductionCallback" function simply prints the result of the reduction and exits.

There is a class being used in the code in figure 3.20 called `CkIndex_Main`. This class is generated by the tools provided by Charm++and used to uniquely identify the various entry methods of the associated class (`Main`, in this case). There are two important points to make here. First, calling an entry method on a `CkIndex_XYZ` class simply returns a unique identifier for the associated entry method in the class `XYZ`. It does not trigger the execution of the entry method. Second, because the code of the entry method is not executed,

```
mainmodule basicReduct_Main {

  mainchare Main {
    entry Main(CkArgMsg*);
    entry [reductiontarget] void reductionCallback(int data);
  };

  extern module basicReduct_Elem;
}
```

```
#ifndef __BASIC_REDUCT_MAIN_H__
#define __BASIC_REDUCT_MAIN_H__

#include "basicReduct_Main.decl.h"

class Main : public CBase_Main {
 public:
  Main(CkMigrateMessage* msg);
  Main(CkArgMsg* msg);
  void reductionCallback(int data);
};

#endif // __BASIC_REDUCT_MAIN_H__
```

Figure 3.19: The main chare class in the “Basic Reduction” program (interface and header files: “basicReduct_Main.ci” and “basicReduct_Main.h”)

```
#include "basicReduct_Main.h"
#include "basicReduct_Elem.h"

#define DEFAULT_NUM_ELEMS  (10)

Main::Main(CkMigrateMessage* msg) { }
Main::Main(CkArgMsg* msg) {

  // Parse the command-line options
  int numElems = DEFAULT_NUM_ELEMS;
  if (msg->argc > 1) { numElems = atoi(msg->argv[1]); }

  // Display a header for the user
  CkPrintf("\"Basic Reduction\" Program\n");
  CkPrintf("  Number of Elements = %d\n", numElems);

  // Create the array of Elems
  CProxy_Elem elems = CProxy_Elem::ckNew(numElems);
  CkCallback *cb = new CkCallback(
      CkReductionTarget(Main, reductionCallback), thisProxy);
  elems.ckSetReductionClient(cb);

  // Start the reduction by having everyone contribute
  elems.doWork();
}

void Main::reductionCallback(int data) {
  CkPrintf("Reduction result: %d\n", data);
  CkExit();
}

#include "basicReduct_Main.def.h"
```

Figure 3.20: “Basic Reduction” Chare Array Program: Main chare class’ source file: “basicReduct_Main.C”

the actual values of the parameters are never evaluated, and thus do not matter. The parameters are only used to identify which version of the function should be called. Just like member functions, entry methods can share the same name if their parameter lists are different. This second point is illustrated in figure 3.20 since the pointer value being passed to `CkIndex_Main::reductionCallback(NULL)`. The parameter can be `NULL` since only the type of the parameter is important. The unique identifier is then passed to the constructor of the `CkCallback` class so that the callback object can identify the exact entry method that is to be called when the callback object is triggered by the runtime system.

Figure 3.21 contains all of the files related to the "Elem" chare array class (header, interface, and source files). For this example, the "Elem" chare array elements are very simple. Each element only perform a single task: when their "doWork()" entry method is instantiated, it contributes a single value to the reduction. The value that each chare array element contribues is its own index ("thisIndex") in the chare array. For an array with N elements, the first element will contribute a 0, the second element will contribute a 1, and so on through the last element which will contribute a value of $N-1$. For this reduction, each contribution is an integer and all of the integers will be summed together. For example, if $N=4$ then the result will be the indices of the array elements added together: 0+1+2+3 = 6. In general, if $N=x$ then the result will be $(N*(N-1))/2$.

3.4.2 Multiple Reductions

[*DMK : Multiple/more complex reduction example to be created.*]

3.5 Multi-dimensional Chare arrays

Chare array can also be multidimensional (from 1D through 6D). Creating and using a multidmensional chare array is basically the same as using a single dimensional array. The main difference is how the "thisIndex" variable is used to index into the multidemensional array. Instead of having only as single value, "thisIndex" has between two and six subcomponents (*(X,Y)*, *(X,Y,Z)*, and so on). A simple 2D 5-point stencil application will be used to illustrate programming with multidimensional chare arrays.

Description of 5-Point Stencil (Jacobi)

The calculation that the 5-point stencil application, also known as a jacobi, uses is fairly straight forward. There is a grid (or matrix) of values. In this example, the grid is 2D. These values typically represent an amount of something at each point in the grid (e.g. heat/temperature). The basic idea is to simulate how these values change over time according to a specified calculation. Time is broken down into decrete timesteps. During each timestep, the value of every element in the grid is updated according to some equation combining one

```
module basicReduct_Elem {

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

```
#ifndef __BASIC_REDUCT_ELEM_H__
#define __BASIC_REDUCT_ELEM_H__

#include "basicReduct_Elem.decl.h"

class Elem : public CBase_Elem {
 public:
  Elem(CkMigrateMessage *msg);
  Elem();
  void doWork();
};

#endif // __BASIC_REDUCT_ELEM_H__
```

```
#include "basicReduct_Elem.h"

Elem::Elem(CkMigrateMessage *msg) { }
Elem::Elem() { }

void Elem::doWork() {
  int value = thisIndex;
  contribute(sizeof(int), &value, CkReduction::sum_int);
}
```

Figure 3.21: “Basic Reduction” Chare Array Program: Elem chare class’ interface and header files: “basicReduct_Elem.ci” and “basicReduct_Elem.h”.

or more values from the previous timestep, typically some function of the values in the surounding elements. For the sake of this example, we will use the average of the current element and the four neighboring elements' values from the previous timestep (hence the name *5-point* stencil).

It is also common to have one or move points in the grid (or borders) held at some fixed value. For example, one could hold one or more elements in the center of the 2D grid at a relatively low fixed value to simulate a cold spot in a temperature simulation. If all of the fixed values do not change in time then eventually the overall simulation will converge to some final state. That is, the magnitude of the largest value change for any of the elements will reduce as time goes on and a stable state will be reached. Once all of the non-fixed values change by an amount less than or equal to a specified error tolerance amount, the simulation is considered to have converged and will exit.

Parallizing 5-Point Stencil

Parallelizing the 5-point stencil application is fairly straight forward. Updating the value of each element during each timestep is based on a fixed computation (average in our case). To parallelize this calculation, we can simply divide the overall 2D grid in both dimensions to create a collection of equally sized tiles. Each tile will contain an equal number of elements from the original grid. We will then map each of the tiles to one of the chare array elements in a 2D chare array, as illustrated by figure 3.22. On the left is the overall grid of values (the data points in the simulation). On the right is the 2D chare array. This two images are overlapped in the center to show how each 2D chare array element will contain a portion of the overall grid (i.e. one tile per array element).

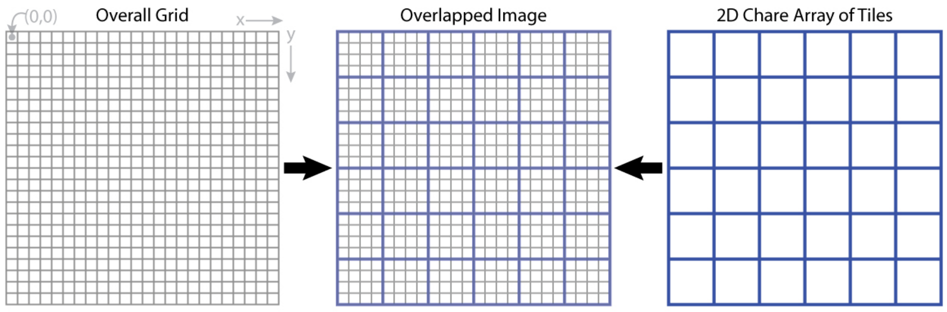

Figure 3.22: Decomposition of 2D jacobi grid into tiles. *[DMK - TODO/FIXME - Update image text to match the text in this example.]*

However, there is a slight complication. Remember that the calculation to update each element's value requires the values of the element's four neighbors. This means that the border elements in each tile will need values from the neighboring tile to complete the average calculation (refer to figure 3.23). Each chare object will have 2D array of elements that represent the its local data. This local data array will have two more elements than required in each dimension. That is, if each tile contains [A x A] elements, then the local data array will be [(A+2) x (A+2)] in size. This allows for a one element border around each tile which will hold *ghost* information from each of the four neighboring tiles[4]. The term ghost data refers to the border data passed by each tile that will be used as input to the element calculations for the neighboring tile's border element's calculations. "Ghost" is used because once the data arrives, it can only be read (i.e. seen), but since the elements don't actually reside there they cannot be written (i.e. touched).

Before a tile can perform its local element calculations, it must first receive neighboring ghost messages and transmit its own ghost data to its neighbors. Each tile sends and receives four ghost messages, one for each neighbor (tiles on the edges exchange ghost data with tiles on the opposite edge). The incoming ghost data is copied into the extra border area in the data array (see figure 3.23). The the actual tile data is in the center [A x A] area of the [(A+2) x (A+2)] data array. Therefor, the ghost data is copied from the edge of the tile data of one tile object to the associated border elements in the data array of the other tile object's data array. Note that this leaves the four corner elements of the data array unused.

Another issue to consider in parallelizing the 5-point stencil application is determing when the simulation is complete. That is, detecting that no non-fixed element changed by more that the error tolerance. To put it another way, that maximum value change seen on any tile was less than a specified error tolerance. Parallelizing this process is a straight forward application of a reduction. Remember that a reduction can preform any commutative and associative mathematical operation. In this case, the mathematical operation of *maximum* has both of these mathematical properties and thus can be used in a reduction to calculate the global maximum value change seen across all the tiles for any given timestep.

Once each tile object has sent its ghost data and received ghost data from all four of its neighbors, the object can proceed to update each of the elements local to the tile object. As it performs the calculation for each element, it needs to keep track of the maximum value change that was seen for the current timestep. Once all of the elements have been updated, the tile object will contribute its maximum value change to a reduction across the entire chare array. Once all of the tile objects in the overall grid have contributed, the result of the reduction will be sent to the main chare object. Based on the global maximum value change, the main chare object will either start another timestep (if the maximum value change is greater than the error tolerance) or cause the application to exit (if the maximum value change is less than or equal to the error tolerance).

[4]For the 5-point stencil example, tiles that are located on the edge of the overall grid will exchange ghost data with tiles on the opposite edge of the grid. As a result, all tile objects will both transmit and recieve four ghost messages, even if they are on the edge of the 2D chare array.

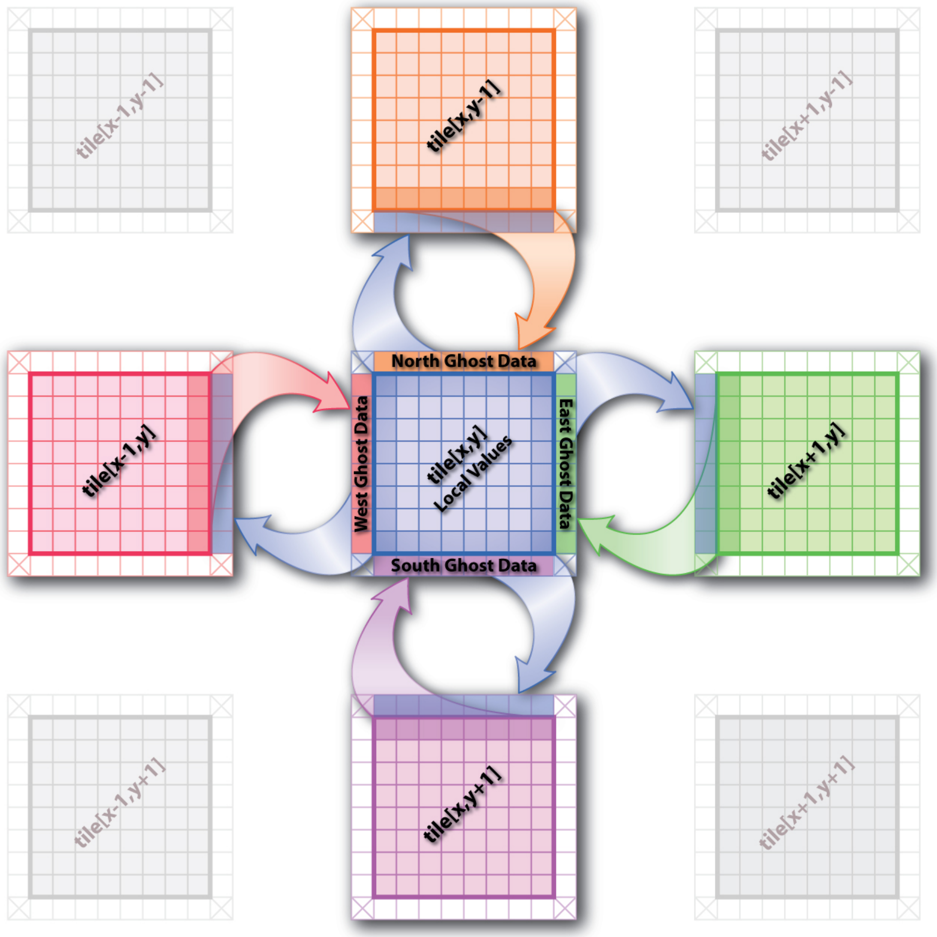

Figure 3.23: Communication pattern of tiles in 2D jacobi program. [*DMK - TODO/FIXME - Update the image text to match the text in this example.*]

Code for Parallel 5-Point Stencil

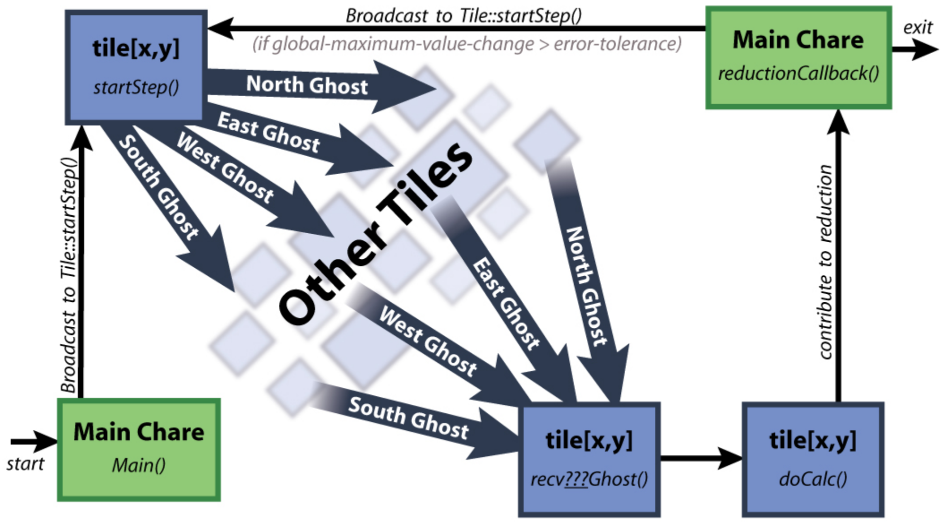

Figure 3.24: Logical flow of 2D 5-point stencil execution from the point of view of a single tile object (*tile[x,y]*). *[DMK - TODO/FIXME - Update image text to match the text in this example.]*

Figure 3.24 show the logical flow of the 5-point stencil application for the point of view of a single chare array element. As with all Charm++applications, the application begins by calling the main chare object's constructor. For reference, figure 3.25 contains the relavant portions of the main chare object's interface and header files.

The source code for the main chare class is located in figure 3.26. The constructor starts by processing the command-line parameters. Once that is complete, it displays a header to the user containing information about the run such as the size of each tile, number of tiles, and so on. The constructor then creates the grid of elements by creating the 2D array of tile objects. The "Main::reductionCallback()" function is set as the reduction client for the chare array and will be called when all of the tile objects contribute to the maximum value change reduction (more about this function later). Finally, the constructor begins the first timestep of the simulation by broadcasting a message to all of the tiles' "Tile::startStep" function.

A broadcast to the "Tile::startStep" function triggers the beginning of a timestep. When invocated, this function sends ghost messages to each of its four neighbors. At the end of the function, there is a call to "Tile::countEvent"; this function call will be discussed shortly. The code for the "Tile::startStep" function is located in figure 3.27.

As a result of all the tiles sending ghost data to each of their neighbors, each tile will also recieve four ghost messages, one from each neighbor. There are four functions that take

```
mainmodule main {

  readonly float targetDiff;
  readonly int gridWidth;
  readonly int gridHeight;
  readonly int tileWidth;
  readonly int tileHeight;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry [reductiontarget] void reductionCallback(float maxStepDiff);
  };

  extern module tile;
}
```

```
class Main : public CBase_Main {
 private:
  CProxy_Tile grid;
  void processCommandLine(const CkArgMsg * const msg);
  void displayUsage(const CkArgMsg * const msg);
  void displayHeader();

 public:
  Main(CkMigrateMessage* msg);
  Main(CkArgMsg* msg);
  void reductionCallback(float maxStepDiff);
};
```

Figure 3.25: The interface and header files for the main chare object in the 2D jacobi application. The interface file also include five read-only variables used by the various chare object.

```
Main::Main(CkArgMsg* msg) {
  processCommandLine(msg);
  displayHeader();
  delete msg;

  // Create the grid tiles and set the reduction client
  grid = CProxy_Tile::ckNew(gridWidth, gridHeight);
  CkCallback* cb = new CkCallback(
    CkReductionTarget(Main, reductionCallback), thisProxy);
  grid.ckSetReductionClient(cb);

  // Start the first step
  grid.startStep();
}
```

Figure 3.26: The main chare object's constructor (entry-point of application).

care of receiving the incoming ghost messages and copying the incoming ghost data into the associated border elements in the data array (remember figure 3.23). Once the incoming ghost data has been copied, each of these functions call the "Tile::countEvent" just like "Tile::startStep" did.

The Tile::countEvent() function, shown in figure 3.29, simply increments a counter and when that counter reaches a certain value (five), it resets the counter and makes a call to Tile::doCalc() which does the actual computation on the elements. This is done to address two issues. First, obviously, the local computation for a tile cannot proceed until all of the ghost messages destined for that tile have arrived since the computation requires the ghost data as input. In essence, these *four events* must occur before the calculations on the tile's elements can begin. Second, before the values of the elements local to this tile can be updated, the values as they were at the start of the timestep must be transmitted to the neighboring tiles before they are modified during this timestep. Otherwise, the neighboring tiles may receive values from the current timestep instead of from the result of the previous timestep. This *event* must also occur before the calculation on this tile can begin. Note that, from the perspective of a single tile, it is possible for the four incoming ghost messages to arrive before the "startStep" messages arrive since there is no guarantee of message ordering in Charm++. That is, the four neighbors could all receive their "startStep" invocations and, in turn, invoke all of their neighbor's receive functions before this tile's "startStep" invocation occurs. Because of this, it is important to make sure that all *five events* occur before a tile proceeds with its own local computation.

```
void Tile::startStep() {
  register const int thisX = thisIndex.x;
  register const int thisY = thisIndex.y;

  // Send to the north (target tile receives from the south)
  float* northGhost = tileData + NORTH_OFFSET + TILE_Y_STEP;
  thisProxy(thisX, thisY > 0 ? thisY - 1 : gridHeight - 1)
    .recvSouthGhost(northGhost, tileWidth);

  // Send to the south (target tile recieves from the north)
  float* southData = tileData + SOUTH_OFFSET - TILE_Y_STEP;
  thisProxy(thisX, thisY < gridHeight - 1 ? thisY + 1 : 0)
    .recvNorthGhost(southData, tileWidth);

  // Send to the west (target tile recieves from the east)
  for (int i = 0; i < tileHeight; i++)
    scratchData[i] =
      tileData[WEST_OFFSET + TILE_X_STEP + (TILE_Y_STEP * i)];
  thisProxy(thisX > 0 ? thisX - 1 : gridWidth - 1, thisY)
    .recvEastGhost(scratchData, tileHeight);

  // Send to the east (target tile recieves from the west)
  for (int i = 0; i < tileHeight; i++)
    scratchData[i] =
      tileData[EAST_OFFSET - TILE_X_STEP + (TILE_Y_STEP * i)];
  thisProxy(thisX < gridWidth - 1 ? thisX + 1 : 0, thisY)
    .recvWestGhost(scratchData, tileHeight);

  countEvent();
}
```

Figure 3.27: The Tile::startStep() function in “tile.C”.

```
void Tile::recvNorthGhost(float* ghostData, int dataLen) {
  memcpy(tileData + NORTH_OFFSET, ghostData,
         dataLen * sizeof(float));
  countEvent();
}

void Tile::recvSouthGhost(float* ghostData, int dataLen) {
  memcpy(tileData + SOUTH_OFFSET, ghostData,
         dataLen * sizeof(float));
  countEvent();
}

void Tile::recvWestGhost(float* ghostData, int dataLen) {
  for (int i = 0; i < tileHeight; i++) {
    tileData[WEST_OFFSET + (TILE_Y_STEP * i)] = ghostData[i];
  }
  countEvent();
}

void Tile::recvEastGhost(float* ghostData, int dataLen) {
  for (int i = 0; i < tileHeight; i++) {
    tileData[EAST_OFFSET + (TILE_Y_STEP * i)] = ghostData[i];
  }
  countEvent();
}
```

Figure 3.28: Entry methods that receive ghost data in the Tile chare class in “tile.C”.

```
void Tile::countEvent() {
  if ((++eventCounter) >= 5) {
    eventCounter = 0;
    doCalc();
  }
}
```

Figure 3.29: The Tile::countEvent() member function (*not an entry method*) in “tile.C”.

Once the counter in the "Tile::countEvent" function reaches five, i.e. all of the *events* have occured so the tile can proceed with the local computation, "Tile::countEvent" makes a call to "Tile::doCalc." Notice that neither "Tile::countEvent" or "Tile::doCalc" are entry methods. Once again, entry methods are just special member functions that can be invoked by other chare objects, or in other words, are the reception points for messages. There is nothing stopping a chare class from having any combination of entry methods and standard member functions.

The "Tile::doCalc" function, which is listed in figure 3.30, does the work of actually updating each element local to the tile object. Since the incoming ghost data is copied into the border elements of the data array, the extra elements on the edges of the data array, the loops that update the elements are striaght forward. The only complication is the starting value for the inner for loop (the x for loop). The code holds the local (0,0) element, actually (1,1) in the data array, of each tile at a constant value of "MAX_VAL." Because of this, the loops need to skip this element and not update it[5]. Therefor the for loops are setup to iterate over all the elements in the tile array, the non-border elements of the data array, except for the single constant element.

The calculation within the for loops is the operation that was previously discussed. The element's current value is averaged with its four neighboring elements' values. As each element is updated, the code keeps track of the maximum absolute value change of the local elements (lines 115-116). Once all of the values have been updated, the tile object contributes its local maximum value change to the global reduction. The result of this reduction is the maximum float value passed by all the tile objects as indicated by the "CkReduction::max_float" parameter passed to the "contribute" call.

Finally, the "Tile::doCalc" function does a small amount of house keeping. Notice that there are two pointers that "Tile::doCalc" operates on: "tileData" and "scratchData". The function only reads from "tileData" and it only writes to "scratchData." At the beginning of each timestep, the buffer pointed to by tileData contains the actually element values for the tile. If a single buffer were used, there were be a data dependency problem. As the for loops started updating the values of some elements (i.e. if line 114 wrote to tileData instead of scratchData), it would change the input values used in future iterations of the loops. Specifically lines 112-113 since they read from *previous* elements in the data array, (x-1,y) and (x,y-1), which may have been written to by earlier iterations of the for loops. To avoid this data dependency, a second data buffer is used for storing the calculated values, "scratchData." Once the for loops have completed, the buffer pointed to by scratch data contains all the updated element values for the tile. The "tileData" and "scratchData"

[5]Notice the call to "Tile::enforceConstants" at the end of Tile::doCalc(). If the for loops were to include these constant elements, the call to Tile::enforceConstants() would overwrite the calculated value for the elements that should remain constant. However, it is important that the for loops in Tile::doCalc() do not include the constant element's value changes in the maximum value change reduction, so the elements still need to be skipped here.

```
void Tile::doCalc() {
  register float maxDiff = ((float)(0.0));

  // Perform the 5-point calc on all elements in the tile
  for (int y = 1; y < tileHeight + 1; y++)
    for (int x = (y == 1 ? 2 : 1); x < tileWidth + 1; x++) {
      float origVal = tileData[XY_TO_I(x,y)];
      float newVal = (origVal + tileData[XY_TO_I(x+1, y)] +
        tileData[XY_TO_I(x-1, y)] + tileData[XY_TO_I(x, y+1)]
        + tileData[XY_TO_I(x, y-1)]) * ((float)(0.2));
      scratchData[XY_TO_I(x,y)] = newVal;
      float diff = fabsf(origVal - newVal);
      maxDiff = fmax(maxDiff, diff);
    }

  // Contribute to the step's reduction
  contribute(sizeof(float), &maxDiff,
    CkReduction::max_float);

  // Swap the data buffer pointers and enforce constants
  register float* tmp = tileData;
  tileData = scratchData;
  scratchData = tmp;
  enforceConstants();
}
```

```
void Tile::enforceConstants() {
  if (tileData != NULL) { tileData[XY_TO_I(1,1)] = MAX_VAL; }
}
```

Figure 3.30: Calculation function and enforce constants function for the Tile class in “Tile.C”.

pointers are then swapped so the “tileData” pointer will once again point to the results of the most recently executed timestep so thos values can be used by the next timestep. The call to “Tile::enforceConstants” enforces the constant values since these elements are skipped by the for loops in “Tile::doCalc” and no assumptions are made about the initial contents of the buffer “scratchData” points to when “Tile::doCalc” begins.

```
void Main::reductionCallback(float maxStepDiff) {
  // Display this step's maximum difference to the user
  static int numStepsCompleted = 0;
  CkPrintf("Step %d: %f\n", ++numStepsCompleted, maxStepDiff);

  // Check to see if the difference is small enough that the
  //   application can complete
  if (maxStepDiff <= targetDiff) {
    CkExit();          // Program finished
  } else {
    grid.startStep();  // Another step needs to be run
  }
}


// Function to process the command-line arguments
void Main::processCommandLine(const CkArgMsg * const msg) {
```

Figure 3.31: The main chare object’s reduction callback function. This function is called when the reduction across the entire grid completes for each step.

Figure 3.31 contains the reduction callback function use by the tile array. First the code retrives the maximum value change from the message and displays the value to the user. Next, the code checks the global maximum value change (i.e. the result of the reduction). If the maximum value change seen across all the tiles is less than or equal to the error tolerance, “targetDiff,” then the simulation is considered to have converged and the application exits. If the maximum value change is larger than the error tolerance, the main chare initiates another timestep by once again invoking the “Tile::startStep” entry method on all the chare array elements (i.e. broadcasting to the chare array).

Figure 3.32 contains the output from an example run of the 2D 5-point stencil program using four processors.

```
$./charmrun +p4 ++local ./jacobi 0.01 10 10 10 10
"2D Jacobi" Program on 4 processor(s)
  Error Tolerance: 0.010000
  Grid Size: [ 10 x 10 ] (in tiles)
  Tile Size: [ 10 x 10 ] (in elements)
Step 1: 0.200000
Step 2: 0.080000
Step 3: 0.048000
Step 4: 0.035200
Step 5: 0.024000
Step 6: 0.021376
Step 7: 0.016845
Step 8: 0.015119
Step 9: 0.012902
Step 10: 0.011600
Step 11: 0.010295
Step 12: 0.009353
```

Figure 3.32: Output of the 2D 5-point stencil program run using four processors.

3.6 Load Balancing

In our next example, we will illustrate how the members of a chare array can be load balanced by the Charm++runtime system. We will not cover the details of how the runtime system does the load balancing (i.e. how it decides which chare objects should be located on what processors, all of the ways in which it can interface with application code, and so on). Instead, we will simply introduce the idea and show a simple example code that triggers load balancing using the "AtSync" method. Many of the details involved with load balancing will be discussed in later in more advanced chapters.

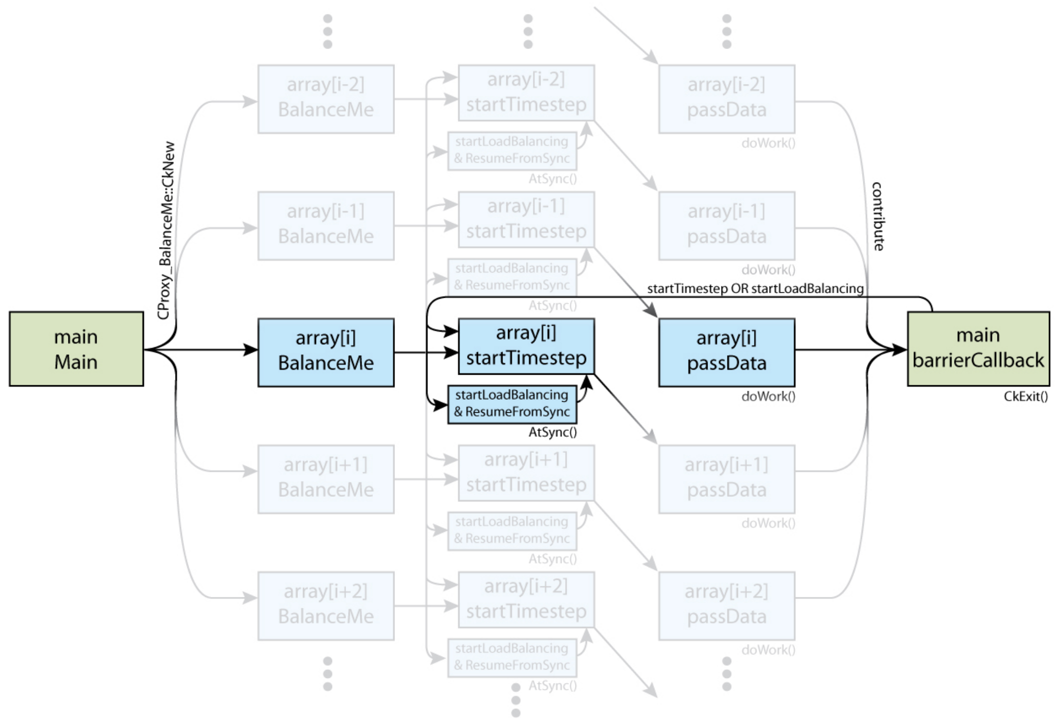

Figure 3.33: Control flow of the example load balancing program.

Figure 3.33 sow the control flow of this application. There are several entry method invocations going on, so it may seem fairly complex at first glance, but it is actually fairly simple. Like all Charm++programs, it begins executing with the main chare object's constructor being invoked by the runtime system. The chare array will be created and the "startTimestep" entry method will be called on each element. This entry method will in turn send some data to the next (i.e. one higher index, except for the last element which will send to index zero). Once the data is received by the previous element via the "passData" entry method, a call to the member function “doWork” will be made. Each element in the chare array will, at creation time, select a random number, “n,” between 1000 and 25000. The value of “n” will indicate *how much work this chare object will do each time “doWork” is called.* With each array element doing a different amount of work, it is possible that some processors will have more work than others, resulting in an overall workload imbalance across the cores. Once the work is complete, the chare array element will make a call to “contribute.” Once all of the array elements have contributed, the reduction will be complete and the callback “Main::barrierCallback” will be triggered. If there are more timesteps to perform, “Main::barrierCallback” will trigger another timestep. Otherwise, it will call “CkExit” to end the program.

A new timestep can be triggered in one of two ways, either with or without also triggering load balancing. Startinga timestep without also triggering load balancing is straight forward (i.e. just broadcast to “array.startTimestep” without doing anything special). However, if we also want have the runtime system load balance the chare objects across the physical processors before starting the next timestep, then we must trigger load balancing. There are multiple ways in which load balancing can occur in Charm++. There will be more about his in later chapters. For now, we will do a simple form of load balancing that requires all the objects involved in the load balancing to come to a synchronization point, “AtSync.” Basically, all of the chare array elements will make a call to “AtSync” to indicate that they are ready to be load balanced. Once all of the array elements have reached this point, the runtime system will take over and load balance the chare array elements using the data that it has been collecting as the program has been executing. For now, we will leave what the runtime system does to load balance the program as a *magic black box* and only focus on how to make use of it. There will be more detailed information about how the runtime system load balancing programs, what data it collects to do so and how in later chapters.

The main chare object’s constructor, located in figure 3.34, will start with the usual housekeeping chores (i.e. read and process command line parameters, initialize readonly variables, initialize it’s own variables, and so on). Once that is complete, it will create a 1D instance of the “BalanceMe” chare array class with length “numElements.”

The constructor of the “BalanceMe” chare array class, located in figure 3.35, starts by setting member variable of its parent class called “usesAtSync.” Setting this variable to true indicates to the runtime system that this chare array element should be load balancined using the “AtSync” mechanism. As we have already stated, there are multiple ways for the load balancing framework to interact with a Charm++program, and this is the method that we will be using for this example. Next we initialize the two member variables, “n” and “myData.” The member variable “n” will be used to indicate the amount of work and data that this chare object will do and send, respectively. The member variable “myData” will simply hold some data (“n” floats to be exact). The values placed in these variables is being arbitrarily calculated since this is just an example program. However, we do want “n” to vary somewhat because the load balancing framework keeps track of both the amount of

```
Main::Main(CkArgMsg *argsMsg) {

  // Initialize variables
  numElements = DEFAULT__NUMBER_OF_CHARE_ARRAY_ELEMENTS;
  numIterations = DEFAULT__NUMBER_OF_ITERATIONS;
  loadBalancingFrequency = DEFAULT__LOAD_BALANCING_FREQUENCY;
  iteration = 0;
  mainProxy = thisProxy;

  // Process command-line arguments
  processCommandLine(argsMsg->argc, argsMsg->argv);
  delete argsMsg;  // Done using the message

  // Create the chare array (NOTE: creation starts first timestep)
  simStartTime = CkWallTimer();
  array = CProxy_BalanceMe::ckNew(numElements);
}
```

Figure 3.34: The constructor for the Main chare class.

```
  // Create a 'random' number that will represent the amount
  //   of work this chare will do
  n = (int)(((float)thisIndex / (float)numElements) * 90000.0f + 10000.0f);

  // Allocate and initialize data array that will hold 'data' to send
  myData = new float[n];
  for (int i = 0; i < n; i++) { myData[i] = thisIndex + i; }

  // Call 'nextStep' on self to start first timestep
  mainProxy.initCheckIn();
}

BalanceMe::~BalanceMe() {
  if (myData != NULL) { delete [] myData; }
}

```

Figure 3.35: The constructor for the BalanceMe chare class.

work performed by each chare and the amount of data sent by each chare (and to which other chares that data was sent). When load balancing is triggered (in this example, when all of the chare array elements have called “AtSync”), this data is passed to a chosen load balancing strategy (more on this later) and that strategy uses the information as it sees fit to determine which chare objects should be migrated. For example, a random load balancing strategy may disregard the load information and simply migrate each of the chare objects to a random physical processor. A greedy load balancing strategy may use the load information to first assign the chare object doing the most work, then the chare object doing the second most work, then the third most, and so on until all of the chare objects have been assigned to a processor. For the sake of explaining the application code, let us not concern ourselves with which load balancing strategy is going to be use (for now).

Finally, the constructor starts the timestep for this chare by calling the “startTimestep” entry method. Notice that this call does not take the form *[proxy].[entry_method]([parameters])*. Since entry methods are also member functions, a given chare object can either call an entry method directly which will treat the call like any other function call in C++, or it can invoke it asynchronously using the proxy object (i.e. invoke it on itself) which will case a message to be generated and the target entry method to be executed at some point in the future (at least after the current entry method completes since only one entry method can be active on any given chare object at a time). In this case, we simply wish to call the “startTimestep” entry method from within this entry method invocation, treating it as a regular C++ function call. [*DMK : TODO : May need to clean this explination up some.*]

```
}

void BalanceMe::passData(int dataLen, float data[]) {

  // Do 'n' microseconds worth of work
  // NOTE: doWork takes seconds and 1000 <= n < 100000
```

Figure 3.36: The BalanceMe::startTimestep entry method.

Figure 3.36 contains the source code for the "BalanceMe::startTimestep" entry method of the "BalanceMe" chare array class. The code for this entry method is quite simple. First, it calculates the index of the *next chare array element*. That is, the chare array element with an index value that is equal to this chare array element's index. Note that the modulus operation is also being applied using the size of the array itself. This will cause the last array element to calculate a "neightborIndex" of zero, causing the last array element to target the first array element. Next, the contents of the this chare's "myData" array are sent to the neighboring chare object via a parameter to the "passData" invocation.

Figure 3.37 contains the source code of the "BalanceMe::passData" entry method and the "BalanceMe::doWork" member function. The actual values of the data being passed to a given chare array element is dismissed (this is just a toy program). Instead, the "passData" entry method calls the "doWork" member function which will loop for "n" microseconds. Note that the value of "n" will be calculated separately for each chare array element and will have a value between 1000 and 25000 (i.e. "doWork" will loop for 1 to 25 milliseconds). Once the "doWork" function has finished looping for the specified amount of time, a "contribute" call will be made. In this case, the value being contributing is inconsequential. We are simply using the reduction mechanism as a way of having each chare array element *check in* with the main chare object. Once all of the array elements have contributed, the timestep is complete and the specified callback function, "Main::barrierCallback," will be triggered by the runtime system.

Figure 3.38 contains the source code of the "Main::barrierCallback" entry method. Again, because we are only using the reduction mechanism as a way of having each chare array element *check in* at the end of a given timestep, we do not care what the result of the reduction actually is and simply disregard it. The "barrierCallback" method then prints a message to the user letting them know that a timestep has completed. Finally, the method decides what to do next: end the program (if all timesteps are complete), start load balancing (if it is time for load balancing to occur), or start a normal timestep (if there are more timesteps

```

  // Contribute to the 'barrier' reduction
  CkCallback cb(CkReductionTarget(Main, barrierCallback), mainProxy);
  contribute(sizeof(int), &workDone, CkReduction::sum_int, cb);
}

int BalanceMe::doWork(double lengthOfWork) {

  // Loop for 'lengthOfWork' seconds
  int workCounter = 0;
  double startTime = CmiWallTimer();
  while ((CmiWallTimer() - startTime) < lengthOfWork) {
    workCounter++;
  }
  return workCounter;  // Returns number of iterations
}

void BalanceMe::startLoadBalancing() {
  AtSync();
}

```

Figure 3.37: The BalanceMe::passData entry method and BalanceMe::doWork member fucntion.

```
void Main::barrierCallback() {
  // Let the user know the iteration has completed
  CkPrintf("Iteration %d completed\n", iteration);

  // Check if this is the last iteration or not.  If so, print
  //   the elapsed time exit.  Otherwise, start the next timestep.
  iteration++;  // NOTE: value is the iteration about to start
  if (iteration >= numIterations) {
    double simStopTime = CkWallTimer();
    CkPrintf("Completed in %lf seconds\n", simStopTime - simStartTime);
    CkExit();
  } else if (iteration % loadBalancingFrequency == 0) {
    CkPrintf("Triggering Load Balancing...\n");
    array.startLoadBalancing();
  } else {
    array.startTimestep();
  }
}

BalanceMe::BalanceMe() {

  // Set the usesAtSync variable (from base class)
  usesAtSync = CmiTrue;
```

Figure 3.38: The Main::barrierCallback entry method.

to do and it's not time to do load balancing yet). Causing the program to exit is done the same way it was in previous programs, simply by calling "CkExit." Starting another normal timestep is also straight forward, invoke the "startTimestep" entry method on each element of the chare array by doing a broadcast to the entire array.

```
  startTimestep();
}

void BalanceMe::pup(PUP::er &p) {
  CBase_BalanceMe::pup(p);  // PUP parent class
  p|n;   // PUP n (size of myData and work length)
  if (p.isUnpacking()) {   // If unpacking, allocate memory
```

Figure 3.39: The BalanceMe::startLoadBalancing entry method and BalanceMe::ResumeFromSync member function.

Every so often (every "loadBalancingFrequency" timesteps), a load balancing timestep needs to be triggered. These timesteps will begin by triggering the load balancer and then continues with a normal timestep once load balancing has been completed. Because we are using the "AtSync" method of triggering the load balancer, the load balancer will only be triggered once all of the chare objects that can be load balanced have make a call to the "AtSync" function. To accomplish this, we call "BalanceMe::startLoadBalancing" at the beginning of the timestep, instead of "BalanceMe::startTimestep." Figure 3.39 contains the code for this function. When the "startLoadBalancing" entry method is called, the chare object it was called on will indicate that it is ready to start load balancing (call to "AtSync") and will go idle (does not invoke other entry methods). Since only the chare array elements are involved in load balancing in this example (recall that we set "usesAtSync" in the "BalanceMe::BalanceMe" constructor), load balancing will start once all of the array elements have made their respective "AtSync" calls. After the runtime system has migrated chare objects across the various physical processors as a result of the load balancing process, each of the chare objects that originally called "AtSync" will have their "ResumeFromSnyc" member functions called by the runtime system. This member function is provided by the base class and can be overloaded by application programmers if an *action* should be taken by a particlar chare object once load balancing is complete. In this example, once the chare array element has been load balanced, it will continue on by starting the next timestep (i.e. "ResumeFromSync" invokes the entry method "startTimestep," also shown in figure 3.39).

At this point, it is natural to ask how does the runtime system migrate chare objects from one processor to another. Again, the details surrounding the load balancing process will be talked about in greater detail in a later chapter. Basically, the Charm++runtime system is using something called Pack-UnPack (PUP) routines to accomplish the object migration. Every data structure that will be passed between chare objects is required to have a PUP routine. The PUP routine is responsible for serializing and deserializing the values contained in a data structure in to and out of a message object, respectively.

```
class Point {

 private:

  float x;
  float y;
  float z;

 public:

  Point(float xi, float yi, float zi) {
    x = xi; y = yi; z = zi;
  }

  float getX() { return x; }
  float getY() { return y; }
  float getZ() { return z; }

  void pup(PUP::er &p) {
    p|x;
    p|y;
    p|z;
  }
};
```

Figure 3.40: A PUP routine for the “Point” class.

Figure 3.40 contains a simple class called “Point” and its PUP routine. We first consider PUP routines in this context to give the reader a simple example to start with. The “Point” class represents a 3D point in space, with “x,” “y,” and “z” values. Except for the “pup” member function, the code should be straight forward. The “pup” routine takes a single argument (“p,” which is a “PUP::er” object passed by reference). We refer to these objects as *PUPers*. There are multiple types of PUPers that are used during the migration processes: sizers, packers, and unpackers. For each type of PUPer, the pipe operator (“—”) is performs a different function. For sizing PUPers, the pipe operator calculates the size of the variable on the right side of the operator and adds that value to an internal counter within the PUPer on the left side of the operator. For packing PUPers, the pipe operator copies the value from the variable on the right side of the operator into a pre-allocated buffer within the PUPer on the left side of the operator. For unpacking PUPers, the pipe operator copies the next value located within a buffer contained in the PUPer on the left side of the pipe operator into the variable on the right side of the PUPer.

```
  }
  PUParray(p,myData, n);  // PUP myData contents
}


#include "ldbDemo1.def.h"

```

Figure 3.41: The BalanceMe::pup routine.

Figure 3.41 contains the code for the PUP routine for the “BalanceMe” chare class. We first introduced the PUP routing for the “Point” class in figure 3.40 beause the PUP routine for the “BalanceMe” class is a bit more complex. First, the “BalanceMe” class has a base class that also has member variables that must be serialized and deserialized as the chare object migrates between physical processors (or, more precisely, between address spaces). As such, the first thing that the “BalanceMe::pup” routine does is call the PUP routine for its parent class. Second, the “BalanceMe” class contains a member variable that is a pointer to dynamically allocated memory, called “myData.” The value of the pointer itself are of no consequence. It mearly represents the location of data that we actually do care about within the current address space. When the data is moved from this address space to another address space as the result of a migration, there is no need for the value of the pointer to remain the same. In fact, ensuring that it would be the same would be somewhat difficult. Instead, we pay attention to the data pointed to by the pointer, an array of float values in this case with the line “PUParray(p, mydData, n);” which PUPs the contents of the array itself. However, before this statement, we have also include an “if” statement. This “if” statement tests whether or not the PUP routine is being called by an unpacking PUP::er. If so, this means that the object data is being deserialized from a buffer within a new address space. Therefore, we must first allocate an array of floats to serve as the memory that will contain the values in the “myData” array within the new address space. After the packing phase is completed in the original address space, the destructor for the chare class will eventually be called, freeing the memory pointed to by “myData” pointer in the original address space. Within the new address space, a special constructor will be called that takes a “CkMigrateMessage” message object. While initialization could be done in this migration constructor, we will not *know* the length of the “myData” array until the member variable “n,” which contains this length, is unpacked within the PUP routine. Regardless, every chare class is required to have this special migration constructor, and thus we include it in this example as well as all of the previous examples, even though we have no use for it in this program.

```
$ ./charmrun +ppn6 ./ldbDemo1 +setcpuaffinity +pemap 0-5 40 4 2
Charm++: standalone mode (not using charmrun)
Charm++: Tracemode Projections enabled.
Charm++> cpu affinity enabled.
Charm++> cpuaffinity PE-core map : 0-5
Charm++> set PE 0 on node 0 to core #0
Charm++> set PE 4 on node 0 to core #4
Charm++> set PE 5 on node 0 to core #5
Charm++> set PE 2 on node 0 to core #2
Charm++> set PE 3 on node 0 to core #3
Charm++> set PE 1 on node 0 to core #1
Charm++> Running on 1 unique compute nodes (8-way SMP).
Charm++> cpu topology info is gathered in 0.001 seconds.
[0] GreedyLB created
Iteration 0 completed
Iteration 1 completed
Triggering Load Balancing...
Iteration 2 completed
Iteration 3 completed
Completed in 1.870763 seconds
Program finished.
```

Figure 3.42: Output from the load balancing demo program.

Now that we have described the code for the load balancing demo program, we will discuss what effects that load balancing has on the performance of the program. Figure 3.42 contains the output from a single execution of the load balancing demo program using six cores. For now, ignore the command line options “+setcpuaffinity” and “+pemap 0-5”. These options are specific to the particular build of the Charm++ runtime system we are using (*multicore-linux64*). Basically, these options just ensure that the threads being used by the runtime system do not continuously switch between the available hardware cores, better utilizing their memory caches. The remaining three parameters to the application, “40 4 2”, indicate the number of “BalanceMe” chare objects in the chare array, the total number of iterations that will be performed, and the number of iterations between attempts to load balance the workload, respectively.

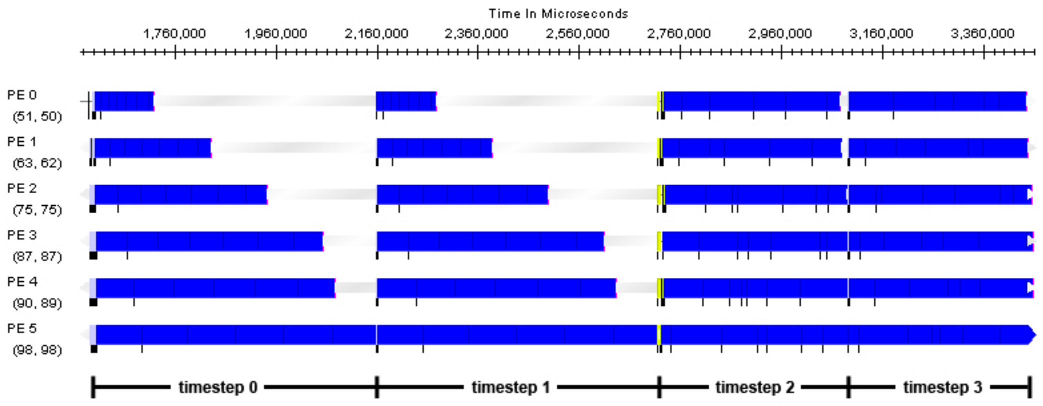

Figure 3.43: Screenshot of a timeline graph showing the execution of four timesteps of the load balancing program executing on six cores.

Figure 3.43 contains a timeline screenshot of the load balancing demo program executing on six cores. The timeline gives the reader an idea of what each of the cores are doing as time passes throughout the execution of the program. We leave a detailed discussion of how to use and interpret timeline graphs, along with other graphs generated using Projections, to a later chapter. For now, we only describe the timeline in the simplest of terms. Each row in the figure represents a hardware core. In this case, we executed the program using six cores as indicated by the "+ppn6" command line option given to "charmrun." Time is increasing from left to right. For the sake of clarity and brevity, we have truncated the beginning of the execution to remove the details of the Charm++ runtime system's initialization (i.e. time does not start at zero). There are many small details that may or may not be clear because of the size of the image. However, the important thing to note for the time being is that each of the larger blocks represents an invocation of the "BalanceMe::passData" entry method. The width of the block represents the amount of time that the entry method took to execute.

Recall that each "BalanceMe" object will perform a different amount of *work*, depending on its index in the chare array. By default, each of the objects in the chare array are assigned to the hardware cores in a block cyclic fashion. That is, the chare objects are divided into groups of contiguous, one group per core. Since the first elements of the chare array do very little work (i.e. smaller values for "BalanceMe::n") and the last elements of the chare array do more work (i.e. larger values for "BalanceMe::n"), this creates an imbalance in the workload assigned to each processing core. The last core will have significantly more work than the first. This can cleary be seen within the first two timesteps in figure **??**. However, once the first two timesteps have completed, each of the chare objects in the chare array will call “AtSync,” causing load balancing to occur. The final two timesteps reflect the effects that load balancing has on the performance in that they take noticably less time to complete. Each of these timesteps takes less time and the blocks are more evenly distributed so that no hardware core become idle waiting for another hardware core to complete its assigned portion of the workload. Using a greedy algorithm (i.e. the algorithm embodied within the “GreedyLB” load balancer that we specified to be used when we compiled the program) along with the PUP routines that we have created, the runtime system automatically migrates the chare objects to balance the workload. Now that we have given the reader a small indication of how the Charm++ runtime system balances workloads, we will end our discussion of load balancing for now. Later chapaters will cover load balancing in greater details, giving the reader a better understanding of the process.

3.7 Summary

We have covered a lot of ground in this chapter, and yet we have only given the reader an introduction on how to make user of chare arrays within Charm++ programs. We have not covered the various options that can be specified at chare array creation time, dynamically adding and removing elements of chare arrays at runtime, *shadow* arrays, and a variety of other topics. At this point, we encourage the reader to try writing a few programs, including the ones presented in this chapter, on their own so the reader might get a more firm grasp on ideas presented here.