A Tutorial Introduction to Charm++

Laxmikant Kale, Nikhil Jain, Jonathan Lifflander
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign

September 3, 2017
Contents

1 Introduction ........................................... 5
   1.1 XMAPP - Concepts ........................................ 8
   1.2 XMAPP Instantiations ..................................... 11
   1.3 Charm++ Features and Benefits .......................... 14

2 Simple Programs and Basic Chares ....................... 21
   2.1 Chares: Concurrent C++ Objects .......................... 21
   2.2 Building, Compiling, and Executing ..................... 24
   2.3 System Utilities: I/O, Program Arguments ............. 26
   2.4 Creating Multiple Chares ................................ 27
   2.5 Chare Proxies and Entry Methods ....................... 31
   2.6 Complete Example: Concurrent File Search ............ 36
   2.7 Overdecomposition and Grainsize ....................... 40
   2.8 Parameters to entry methods: Arrays and other types ... 43
   2.9 Exercises ............................................... 43

3 Chare Arrays: Indexed Collections of Chares ........... 47
   3.1 A Single Ring ........................................... 48
   3.2 Multiple Rings .......................................... 52
   3.3 Reductions ................................................ 60
   3.4 Prefix Sum with Recursive Doubling .................... 65
   3.5 Multidimensional Chare arrays .......................... 69
   3.6 Exercises ............................................... 79

4 Structured Dagger (SDAG) ............................... 83
   4.1 Traffic sim using SDAG: no barrier ..................... 83
   4.2 SDAG constructs: when, for(all), overlap, case, and others ... 83
   4.3 Exercises: fibonacci, prefix sum ...................... 83
   4.4 Introduction ............................................ 83

3
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Specifying Dependencies in SDAG: the when Clause</td>
<td>86</td>
</tr>
<tr>
<td>4.6</td>
<td>Blocks of C++ Code: the serial Clause</td>
<td>87</td>
</tr>
<tr>
<td>4.7</td>
<td>Other SDAG Constructs</td>
<td>87</td>
</tr>
<tr>
<td>4.8</td>
<td>Example: Parallel Prefix Sum</td>
<td>90</td>
</tr>
<tr>
<td>4.9</td>
<td>Enhancing Concurrency</td>
<td>93</td>
</tr>
<tr>
<td>4.10</td>
<td>More Constructs</td>
<td>95</td>
</tr>
<tr>
<td>4.11</td>
<td>Example: 5-point Stencil</td>
<td>95</td>
</tr>
<tr>
<td>5</td>
<td>Load Balancing</td>
<td>99</td>
</tr>
<tr>
<td>5.1</td>
<td>Migration-based load balancing</td>
<td>100</td>
</tr>
<tr>
<td>5.2</td>
<td>PUP Framework</td>
<td>100</td>
</tr>
<tr>
<td>5.3</td>
<td>SeaLife</td>
<td>103</td>
</tr>
<tr>
<td>6</td>
<td>Commonly Used Constructs</td>
<td>111</td>
</tr>
<tr>
<td>6.1</td>
<td>Callback</td>
<td>111</td>
</tr>
<tr>
<td>6.2</td>
<td>Messages</td>
<td>112</td>
</tr>
<tr>
<td>6.3</td>
<td>Priorities</td>
<td>118</td>
</tr>
<tr>
<td>6.4</td>
<td>Quiescence Detection</td>
<td>118</td>
</tr>
<tr>
<td>6.5</td>
<td>Dynamic insertion, deletion, on demand</td>
<td>119</td>
</tr>
<tr>
<td>6.6</td>
<td>Sections and Delegation</td>
<td>119</td>
</tr>
<tr>
<td>7</td>
<td>Threaded Entry Methods</td>
<td>123</td>
</tr>
<tr>
<td>7.1</td>
<td>Threaded and Sync Methods</td>
<td>123</td>
</tr>
<tr>
<td>7.2</td>
<td>A simple illustrative example of blocking invocation</td>
<td>124</td>
</tr>
<tr>
<td>7.3</td>
<td>Futures in Threaded Entry Methods</td>
<td>127</td>
</tr>
<tr>
<td>7.4</td>
<td>Callback for resuming a thread</td>
<td>132</td>
</tr>
<tr>
<td>7.5</td>
<td>When not to use threads</td>
<td>134</td>
</tr>
<tr>
<td>7.6</td>
<td>Exercises</td>
<td>135</td>
</tr>
<tr>
<td>8</td>
<td>Acknowledging the Physical: Groups, Nodegroups, Maps, etc.</td>
<td>139</td>
</tr>
<tr>
<td>8.1</td>
<td>Charm++ view of the physical machine</td>
<td>139</td>
</tr>
<tr>
<td>8.2</td>
<td>Array maps</td>
<td>140</td>
</tr>
<tr>
<td>8.3</td>
<td>Groups</td>
<td>142</td>
</tr>
<tr>
<td>8.4</td>
<td>Node-Groups and Effectively Utilizing Shared Memory</td>
<td>143</td>
</tr>
<tr>
<td>9</td>
<td>SHM: This definitely needs a chapter of its own</td>
<td>145</td>
</tr>
<tr>
<td>9.1</td>
<td>Execution in Shared Memory Mode</td>
<td>145</td>
</tr>
<tr>
<td>9.2</td>
<td>Readonly Variables</td>
<td>146</td>
</tr>
<tr>
<td>9.3</td>
<td>Conditional Packing</td>
<td>146</td>
</tr>
<tr>
<td>9.4</td>
<td>NodeGroups</td>
<td>150</td>
</tr>
</tbody>
</table>
10 Designing Parallel Programs with Charm++ 151
  10.1 Designing a Charm++ Application ................................. 151
  10.2 General Principles of Designing Charm++ Application .......... 154
  10.3 Lennard-Jones Molecular Dynamics ................................. 154
  10.4 Prelude: LiveViz: 2-D Wave Equation on Structured Grid .... 155

11 Advanced Topics 165
  11.1 Modules .............................................................. 165
  11.2 Threads .............................................................. 165
  11.3 Futures .............................................................. 165
  11.4 Readonly .............................................................. 165
  11.5 CkLoop ............................................................... 165
  11.6 Entry method attr .................................................. 165
  11.7 Tracing options ..................................................... 165
  11.8 Adv LDB .............................................................. 165
  11.9 Dynamic Insertion .................................................. 165

12 Performance Analysis with Projections 167
  12.1 Loading logs and Range selection ................................. 168
  12.2 Aggregate Views .................................................... 168

13 Checkpointing 173
  13.1 Saving the State of an Application .............................. 173
  13.2 Fault Tolerance ...................................................... 178
Chapter 1

Introduction

Since the advent of digital computers in the middle of the last century, users have continually hungered for faster computers. They have marveled at the speeds of the available computers, developed ways to use that speed to solve their problems faster, to find better solutions to their problems, and to enlarge the scope of solvable problems. Then, they aspired to solve larger, more complex problems and subsequently desired more powerful computers with which to solve them.

The demand for computational power has been continuously met by rapid advances in hardware technology and microprocessor architectures. Historically, most of these advances have come from the ability to produce smaller electronic circuits on silicon chips. The smaller the circuit can be made, the faster it was and the less expensive it was to produce. Together, these technological advances led to the phenomenon of declining prices with dramatically increased performance. Thus the inexpensive personal computers sitting on desktops in homes are millions of times faster than the earliest computers, yet thousands of times less expensive than them.

However, this convenient trend could not continue forever. Indeed, it is expected that the current semiconductor technology will continue to shrink the size of circuits until circa 2020 or so, to the point where one may have between 30 to 50 billion transistors on a single chip, in contrast to about 5-7 billion now. However, the speed as represented by the clock frequency has already stopped increasing since about 2004, because chips will get too hot with higher frequencies.

Parallel computing is an approach that helps us evade this problem and will allow the trend in speed improvements to continue for years to come. The idea is simple: to solve a computational problem faster, use many processors to work on different parts of the problem. However, anyone who has tried reduce the completion time of a task by employing more people knows that this involves complex coordination problems that affect how efficiently speed-ups can be obtained. To facilitate this, we first need an infrastructure that will allow
multiple processors to communicate. Such infrastructures are provided by parallel computers. Starting circa 1985, many commercial parallel computers became available. In 1994, there were computers available with tens, hundreds, or even thousands of processors each. Most of them employed state of the art microprocessors, each capable of hundreds of millions of operations per second. As we write this, in early 2016, almost all the computers in the list of top 500 most powerful computers have at least 8,000 processor cores each capable of billions of operations per second. Today, the machine with most cores has almost 1,600,000 cores! Here, the number of cores in a computer corresponds to the number of separate streams of parallel computations.

In addition to the phenomenal increase in the power of the top supercomputers, the other interesting story of this era is the rising ubiquity of smaller clusters of 100 cores or more. With 4-8 cores per chip (called “socket”) and 2-4 sockets per server board, it is easy to put a 100 core machine with 8-16 servers. The costs are not much more than what an engineering workstation used to cost in the 1990s. Thus many department-size units can now afford parallel computers with significant power, and — if only they could exploit them with effective parallel applications — can use computational analysis to improve their products. This is likely to cause a significant impact in various segments of the industry and academic research, in the coming decades.

To date, there are many applications that can benefit from such small and large machines. These include complex applications, such as global weather forecasting, that can use thousands of processors, and simpler applications, such as text processing, that can benefit from the speed-ups possible with just a few processors. There are emerging applications in multimedia, online transaction processing, and decision support systems that “mine” and analyze huge amounts of data.

Along with the growth in the number of cores and the power of systems, several methodologies to program these systems have also been developed. This book presents one such effective methodology for parallel programming, which has been proven successful in a variety of contexts over the last 25 years. This methodology is embodied in a parallel programming model called XMAPP.

Broadly speaking, XMAPP is a paradigm for describing parallel interactions and runtime behavior only. It does not take any position on how sequential components of a parallel program are expressed. They can be expressed using popular languages like C, C++, and Fortran, or via a newly defined language. In Section 1.2, we will briefly introduce several instantiations of XMAPP: Charm++, AMPI, Charj, etc. Beyond that, this book will primarily focus on Charm++, which is the most widely-used implementation of this model. We begin with an introduction to the XMAPP programming model.
1.0.1 Alternate Introduction

ALTERNATE INTRODUCTION (partially written: either merge with above, or expand it. Or it could go into foreward.

It was sometime around 1978. I had just joined graduate school at Indian Institute of science in Bangalore for a Masters degree. Prof. William Wolf, then at Carnegie Mellon University, was visiting, and gave a lecture on C.MMP and Hydra. These were his projects in parallel computing. It was then that I learned about parallel computing for the first time. They had put together 16 PDP 11 computers in a network to make a parallel computer. To motivate the need for parallel computing he argued that although sequential individual processors were getting faster every year, this growth cannot continue for a long time. After all there were physical limits to how small you can make circuits. So, he said, when it runs out of steam, you got to have parallel computing to keep making faster computers. Users, especially those doing engineering and science simulation (in those days), always need faster processors to do cutting edge research.

This was one of the two seminars I attended that induced me to pursue a doctorate in parallel computing in the USA, and eventually, to pursue parallel computing. I was to hear this rationale for parallel computing repeatedly over the years. Yet, the silicon engineers the parallel computing community a run for its money. They kept making single computer faster, year after year, for 30+ years after that (and kept the well-known Moores law going for 40+ years). The frequencies (and therefore raw single-thread speed, to a large extent) stopped increasing around 2003. And the vaunted Moores law itself, about continuous increase in the number of transistors per chip, is finally about to end, this time indisputably. However, the other part of the motivation given by Prof. Wolf and others, remained valid all through. (CHECK if his papers refer to case applications)

Users are always looking for faster computers. To use a phrase used by many leaders in parallel computing, a supercomputer was like having access to time travel. Even during the heyday of Moores law, you could use a parallel supercomputer of the day to compute at a rate that would be possible only 10-15 years later on a desktop or engineering workstation. That time travel was of strategic use in industry or in science, especially for governments, which made it worthwhile to spend tens of millions of dollars on it.

After 2003, as more and more transistors were added to a chip by Moores law (i.e. by decreasing size of the transistors), the only remaining reasonable way of using them was to put multiple processor cores on a chip. This made parallel processing ubiquitously available, as well as inescapable for performance-oriented applications. Of course, as application developers and users strove for strategic advantage over competitors and for breakthroughs in science and engineering, distributed memory computing became attractive at various price-points, including small clusters 8-32 nodes.

End Alternate Introduction
1.1 XMAPP - Concepts

One of the basic premises of XMAPP is that although parallel computers are built on a variety of different architectures, a simple cost model unifies them all; this cost model provides an effective design principle for writing parallel programs. The cost model is based on the fact that processors can access local data much faster than remote data. Here, local data means data that resides in a processor’s cache or its private memory. Remote data is all other data; on some machines this may be data in another machine’s private memory while on others it may be data in global shared memory modules. A programming model that exposes this cost model and encourages writing programs that respect locality of data access is likely to lead to efficient programs.

The other foundational principle of the XMAPP model is to strive for an optimal division of labor between the programmer and the system. Here, the “the system” is loosely defined as the part of the programming system which performs any work that is not directly the work of the user program. This potentially includes a compiler and the runtime system.

When a parallel application is developed, the following decisions must be made:

1. How are the data and computations partitioned/decomposed?

2. Where are the resultant partitions and computations mapped to hardware resources (such as nodes and processor cores)?

3. In what sequence are computations executed on each such resource?

In traditional parallel programming approaches for distributed-memory machines, such as MPI, all these decisions are made by the programmer. This makes writing parallel programs complex, especially when the program exhibits dynamic behavior. Parallelizing compilers aim at automating all three decisions, but are often unsuccessful in extracting adequate parallelism from the sequential program. The idea behind the XMAPP model is to find a ideal division of labor between the runtime system and programmer by automating steps 2 and 3, but leaving decomposition to the programmer. The programmer has knowledge of the application domain, and hence is best suited for dividing the work into parallelizable units. On the other hand, the runtime system has knowledge of the architectural parameters and has the ability to introspect the application’s execution on that platform. This makes it an ideal candidate for mapping and scheduling work units to hardware resources.

To materialize the above mentioned principles, XMAPP combines three key ideas: overdecomposition, asynchronous message-driven execution, and migratability. These components complement each other, enabling parallel interactions to be described effectively. The synergy between them, supported by a powerful runtime system, leads to highly-efficient applications and high productivity for the programmer.
1.1. **XMAPP - CONCEPTS**

### 1.1.1 Overdecomposition

In traditional models such as MPI, the decomposition of the application domain is intrinsically tied to the number of hardware resources. In other words, the application is programmed to a specific hardware configuration. This methodology has two distinct disadvantages. First, the natural expression of the application domain is often not in terms of the number of hardware units, leading to a loss in programmer productivity when programming for specific hardware configuration. Second, if the application exhibits any dynamic behavior, tying the decomposition to the hardware necessarily limits how the runtime system can adapt to these changes.

The XMAPP methodology is to decompose the application in a natural way that expresses more parallelism than the available hardware units. To be clear, we are not trying to express the maximum available parallelism because this often leads to performance problems due to small work unit sizes that cannot be efficiently executed. Instead, we advocate decomposing the application into medium-sized work units that are large enough to enable efficient execution, while being sufficiently abundant so the runtime system can rebalance work as necessary. The important point is to liberate decomposition from the number of hardware resources such as processors, which results in a user’s view of application as shown in Figure 1.1 (left). Note that it is not enough that there are many work units on each processor. XMAPP requires that the program be expressed from each unit’s point of view. No separate user-written code coordinates activities of all work units assigned to each processor.

In this book, we often use the term “processor” to stand for independent hardware resources such as a cores, hardware threads or even nodes, when we do not need to be specific about which of these units are meant. Similarly, we use the phrase “work units” for simplicity, but some of the units may be pure “data units”, whose work is simply to serve the data, or to update it, when asked.

This methodology has several benefits. First, when each processor has multiple work units mapped to it, computation and communication can be overlapped: while one work unit is waiting for data to arrive, another work unit can execute. Second, work units can be scheduled (e.g. some work units prioritized over others) by the runtime system based on application-specific information provided by the programmer and observations made by the runtime system. Third, divorcing the computation from the hardware enables modularity, allowing parallel modules (that may interact) to be interleaved and partitioned in an efficient manner by the runtime system.

### 1.1.2 Asynchronous Message-Driven Execution

Work units in XMAPP interact with each other asynchronously. For intuitiveness, we will use the word “message” for any interaction between work units. When a work unit (A) sends a message to another work unit (B), A does not wait for B to return any value. If B wants
Figure 1.1: Overdecomposition: User’s view (left figure) of the program does not involve processors. The runtime system maps the work units to processors and schedules their execution (right figure).

to a send a result based on the first message, it sends another asynchronous message to A. To explain the difference between asynchronous communication in XMAPP from synchronous communication such as a function call, consider the following analogy. A public or private function call is like a telephone call: when the caller communicates with a recipient and asks a question, the caller waits for a reply and does not continue with its work until it receives a reply. Asynchronous communication, on the other hand, is like a mail message: the caller sends some information and possible instructions, and continues with its work without waiting for a reply. The recipient acts on the mail message at its leisure. It may collect more information before its respond to the caller. It may even delegate the job of responding to another entity. In some cases, the caller may not need any response from the recipient.

This model of communication, which is distinct from a synchronous RPC (remote procedure call), is common to many parallel systems. But combined with overdecomposition, it leads to a profound sense of asynchrony, as explained next. After a problem has been overdecomposed into work units that communicate asynchronously by the programmer, the work units are been assigned to processors by the runtime system, thus creating the runtime system’s view of the application as shown in Figure 1.1 (right). The next important decision is determining the execution order of work units on the processors. In the XMAPP model, the flow of messages between work units drives the execution on each processor. In this
1.2. XMAPP Instantiations

style of execution, called message-driven execution, messages from one work unit to another determine the action that the recipient work unit performs. When a message is received, it is stored in a pool or queue and scheduled on that resource by the runtime system (see Figure 1.1). So, what a processor does next (after it finishes the computation in a work unit triggered by a message) is not explicitly specified by the programmer, but rather decided by what message is at the head of the scheduler’s queue.

While overdecomposition provides an opportunity for overlap between communication and computation, asynchronous message-driven execution enables XMAPP to exploit this overlap. Asynchrony decouples the execution of work units to some extent, and allows useful computation to be performed while communication happens in the background. Message driven execution complements asynchrony by enforcing a need based execution that helps runtime system to identify computations that are ready to be performed.

1.1.3 Migratibility

The third and final key component of the XMAPP model is migratibility: work units are flexible in terms of placement. Since the programmer describes the parallel interactions in terms of the interacting work units instead of processors, the runtime system can change the mapping of work units to the processors and migrate them during the execution as it sees fit. This does not affect the execution of the program since the runtime system is aware of the location of the work units, and hence delivers the messages targeted at work units to the right processors. Migratibility enables a vast array of optimizations, ranging from automatic load balancing to fault tolerance. For example, based on the interactions the runtime observes during the application’s execution, the runtime system can optimize the placement of the work units. If two work units communicate heavily throughout the execution, the runtime system can co-locate them to reduce communication (perhaps so they share a memory domain). When that’s not possible, it may place them on nearby nodes in order to minimize communication contention.

1.2 XMAPP Instantiations

The XMAPP model has been implemented in a variety of different languages ranging from Charm++, the widely-used C++-based implementation that this book will focus on, to AMPI (Adaptive MPI), which allows MPI programs to take advantage of overdecomposition and migratibility. In addition, the XMAPP model has been implemented in several domain-specific languages (Multi-phased Shared Arrays, Structured Dagger, Charisma, etc.) that provide powerful abstractions for expressing certain types of parallel interactions. Here, we briefly introduce the two most commonly used languages: Charm++ and AMPI.
CHAPTER 1. INTRODUCTION

1.2.1 Charm++

Charm++ is a parallel language and runtime system implemented in C++, in which an application domain is decomposed into C++ objects. Some of these objects, called chares as shown in Figure 1.2, are managed by the runtime system.

Chares: Respect for data locality is essential for designing efficient parallel programs. In sequential programs, an object captures the notion of locality well: it consists of some data and a collection of methods that can update this data. An object encapsulates its data in the sense that the data can be accessed or modified only through the methods provided by the object. In parallel programming, objects also express locality in an additional context. Any data that is within the object can be assumed to be local data, which can be accessed efficiently. Data from another object may or may not be accessible quickly depending on the location of that object. Chares, specially designated objects in Charm++, build on this abstraction in Charm++ and enables invocation of methods on remote objects.

A chare is a C++ object which also has some additional properties. To make the programmer aware of the potential cost of interacting with remote objects, we must distinguish calls to methods of local objects from calls to methods of remote objects. For this reason, a chare consists of some local data, private methods that can be called only from within other functions in the same chare, and specially designated entry methods that can be called by any remote object that has a proxy to that chare. A chare’s proxy is a location-oblivious...
1.2. XMAPP INSTANTIATIONS

handle, i.e. a proxy is the equivalent of a reference/pointer to a chare object, but one that works even from remote nodes, and allows entry methods to be invoked on a possibly-remote chare (details of these concepts will be provided in Chapter 2).

New chares can be created from any chare in a Charm++ program. Creating a chare is tantamount to creating a new piece of work. The system assigns this piece of work to some processor in accordance to its dynamic load balancing strategy. This assignment does not necessarily happen at the same time as object creation. The creation call immediately returns after depositing a seed for the new chare with the system. All that is guaranteed is that eventually, on some processor, that chare will be created and will execute its initial constructor call using the arguments provided. In fact, all system calls in Charm++ are non-blocking: they do not wait for actions on remote processors.

To invoke an entry method of an existing chare, a proxy to that chare must be obtained. When you create a chare, its proxy is returned to you; Alternatively a chare can find its own proxy by making a system call and this proxy can be passed to other chares.

**Chare Arrays:** A *chare array* is a collection of chares with a global name for the collection. Each element of the collection is identified by an index. The index may be an integer, but in general, may be one of many index types. A chare array may be multi-dimensional, sparse or dense, and may have elements inserted or deleted dynamically at runtime. For example, a one dimensional chare array, represented by its global proxy A, may have members identified by indices 0..99. However, even within one-dimensional arrays, chare arrays may have non-contiguous indices (e.g. an array consisting of 1000 elements with indices ranging between 1 billion and 2 billion). Other examples of index types include 2-D, 3-D, and higher-dimensional integer indices, or bit-vector indices that represent a position in an oct-tree, etc. There can be multiple chare arrays of different types in a single application.

It is important to remember that a chare array does not necessarily represent an array of basic types. Each element of an array is a full-fledged medium-grained chare object that may hold whatever data the application may choose to assign to it: linked lists of particles, arrays of grid points, finite-element subgrids, or trees, for instance). Each element has entry methods that other chares invoke, and a life-cycle of its own.

A program may invoke an entry method on an individual element of a chare array, or broadcast a method invocation to all its members. The chare array elements may also participate in reduction operations (e.g. adding up data across all its members). The important feature of chare arrays is that the elements of an array may be migrated across the processors by the runtime system. Yet, the application does not need to be aware of the location of each array element. When a method is invoked on a certain index of the chare array, the system will find where this elements lives, and deliver the method invocation to it, even when the element has recently migrated to a different processor.

---

For an exception, see a later Chapter 11, but we prefer that a good Charm++ programmer will avoid using those techniques and use them only for exceptional circumstances.
1.2.2 Adaptive MPI

Adaptive MPI (AMPI) is an implementation and an extension of the MPI standard on top of the Charm++ runtime system (RTS). It is a powerful interface that enables legacy code written in MPI to take advantage of XMAPP model of parallel programming. AMPI implements process virtualization by mapping multiple MPI ranks to every physical core, which in turn are managed by the Charm++ RTS. Each MPI rank is executed using a user-level thread, which enables the RTS to switch among different ranks with low overheads. Any legacy MPI code can be run using AMPI without any changes (except for eliminating global variables), thereby gaining the added benefits of overdecomposition and asynchronous message-driven execution. AMPI also extends the MPI standard to support migratability, thereby implementing all the components expressed in XMAPP.

1.3 Charm++ Features and Benefits

Before we discuss the benefits of Charm++ and other XMAPP languages, we briefly discuss how Charm++ and its RTS is implemented. This will hopefully be useful in understanding the reasons for the listed benefits.

1.3.1 Under the hood: The Execution Model of Charm++

As stated earlier, each chare in Charm++ is a special C++ object, distinguished from plain C++ objects by the fact that some of its member functions are designated as “entry” methods. Entry methods can be invoked asynchronously from objects that may be on other processors. For example, \( A[i].\text{foo}() \) specifies that the method, \( \text{foo} \), should be invoked on the \( i^{th} \) element of the collection named \( A \), where \( A \) is technically the name of the proxy to the collection. What actually happens when such a method invocation is called is shown in Figure 1.3. The figure shows two processors, each with several chares. Each processor also has a pool of “messages”. Each message is a pending method invocation meant for chare(s) on that processor. Figure 1.3 also shows a scheduler on each processor whose job it is to select a method invocation from the pool to execute next.

On processor 0, the calling chare makes the invocation \( A[i].\text{foo}() \). When such a call is made using a proxy, the runtime system packs the parameters passed to the call into a contiguous (“serialized”) message. This message also encodes the method name and the destination chare’s identification. The runtime then figures out, using an efficient location management service, the processor on which the destination chare is located, and sends the message towards that processor. At this point, control returns to the calling chare on processor 0 (the sender processor), which continues its execution.

At some point in future the message would be delivered on the destination processor (in this case processor 1), and enqueued in the pool of messages on that processor. Later
1.3. CHARM++ FEATURES AND BENEFITS

Figure 1.3: Mechanics of method invocation in Charm++: user’s call to a proxy is translated to a message that results in invocation of the method on the object eventually.

on, the scheduler on processor 1 selects this message, identifies the chare it is targeted at, makes sure that the chare hasn’t migrated away, unpacks the message to reconstruct the arguments to foo, and invokes the method foo on that chare. This method execution may end up creating messages that are sent on their way as described above. When this method completes execution, control returns to the scheduler, which then proceeds to select another message from its pool.

During the execution, if it so desires, the RTS can migrate a chare from one processor to another processor. The location management will ensure that messages will be correctly delivered to the chares; the other operations on chare arrays (e.g. broadcasts) are implemented so that they can handle such migrations.

Note that the RTS is involved in scheduling each entry method, so it knows the amount of computation ascribed to each chare. Similarly, the RTS mediates communication between chares, so it can keep track of which chare communicates with which other chares, and with what size messages. XMAPP’s model based on over-decomposition and migratability thus imparts the ability to do “introspection” to the RTS. Adding adaptivity to introspection creates a powerful adaptive runtime system (see Figure 1.4). Adaptivity here refers to the ability of the RTS to effect optimizations by using mechanisms at its disposal such as chare migration, execution sequencing, and communication operations.
1.3.2 Benefits

**Prediction**: One of the direct benefits of the RTS driven approach is the *ability to predict data access patterns accurately*. The scheduler can peek at its queue and know what the next few method invocations are. It knows which chares are involved in those; thus, for example, it can prefetch data for those chares from lower-level memory to higher level ones, such as the scratchpad memory in some modern accelerators. In comparison to the principle of locality, this prediction is much more accurate. And yet, an entire industry and architecture was built using the principle of locality – the idea of cache hierarchies. So, this predictability of access in Charm++ is likely to be of increasing use in future architectures.

**Communication**: The prevalent methods for parallel programming typically leads to a compute-communicate-and-repeat pattern, as shown in Figure 1.5. One may optimize communication to reduce it to say 20%, but that results in the network being idle for 80% of the time! Machine vendors are often asked to build faster networks so as to reduce communication time, or at least to not increase it since the next generation nodes are likely to do more computation per unit time.

With Charm++, as with other XMAPP languages, there are many chares per processor and *the communication is spread over the entire iteration*, as shown in Figure 1.6. This leads to a much better utilization of the communication resources. Further, communication and computation are overlapped without any extra effort by the programmer.

**Decomposition**: As stated earlier, forcing a decomposition tied to the volume of hardware resources may negatively impact productivity as often applications are better expressed with a domain-based decomposition. XMAPP languages *enables such decompositions independent of the number of processors*. Another negative impact of hardware bound decomposition is the arbitrary placement of possibly unrelated domain-specific entities on a processor. For
1.3. CHARM++ FEATURES AND BENEFITS

• Current use of communication network:
  – Compute-communicate cycles in typical MPI apps
  – So, the network is used for a fraction of time, and is on the critical path

• So, current communication networks are over-engineered for by necessity

P1

P2

BSP based application

Figure 1.5: An example of compute-communicate pattern in bulk-synchronous parallel programs.

Overdecomposition enables overlap

P1

P2

Figure 1.6: Example to demonstrate spread of communication and overlap of computation and communication in Charm++.

example, Figure 1.7 shows a typical decomposition performed in an application for rocket simulation. Solids and Fluids are two important, but loosely connected, entities that constitute the physical space being simulated. Due to the requirement to decompose these entities among \( P \) processors, we observe that arbitrary partitions that may not be correlated are placed on a processor. In contrast, Figure 1.8 presents a domain-based decomposition enabled by XMAPP. Each of the important, Solids and Fluids in this case, are independently decomposed into a number of partitions (chares) suitable for them. The RTS places the chares in a predefined manner, and observes as the execution proceeds. Later on, if needed, the RTS relocates the chares in order to optimize for computation balance and communication overhead.

**Compositionality**: Scheduling of the chares by the RTS enables XMAPP languages support compositionality of independent (and dependent) modules. Essentially, the RTS schedules executions of various chares from different modules such that overheads of one module, e.g. communication delay, can be utilized for progressing useful computation in other modules as shown in Figure 1.9. In contrast, for hardware based decomposition, such compositionality is not achievable. In such cases, different modules can only be executed by either a division of processors among modules or by a time-quantized utilization the processors.

**Prioritized scheduling**: Scheduling by the RTS is further augmented by adding support for
Decomposition Independent of numCores

Rocket simulation under traditional MPI

Solid
Fluid

1
2

. . .

Solid
Fluid

P

Figure 1.7: Hardware-bound decomposition: irrespective of domain requirements, different entities are divided among \( P \) processors, and are arbitrary placed with one another.

Solid\(_1\)
Solid\(_2\)
Solid\(_3\)
Solid\(_n\)

Fluid\(_1\)
Fluid\(_2\)
Fluid\(_m\)

Figure 1.8: Domain driven decomposition in XMAPP: based on the type of entities, they are decomposed differently; the RTS observes these decompositions and place them intelligently.

Figure 1.9: RTS controlled message-driven execution enables progress of many independent modules, essentially overlapping overheads of one with useful work of others (left). On the right, an example of time division that will be required in hardware-bound decomposition.
different queuing strategies based on priorities. The programmer assigns priorities to entry methods that are reflected in associated messages that leads to their invocation. The RTS uses these priorities to reorder scheduling on a processor.

**Automatic, dynamic load balancing:** Given its responsibilities and capabilities, the Charm++ runtime system supports measurement-based load balancing: it automatically instruments every chare’s communication and computation, and uses this information to redistribute chares. The Charm++ runtime system includes a suite of load balancers, including centralized, hierarchical, and distributed strategies that can selected using a simple command-line argument and used in a application-oblivious manner.

**Automatic checkpointing and restart:** Charm++ runtime system uses the migratability of chares to support checkpoint/restart. The state of a chare can be checkpointed to disk or another processor’s memory automatically using a simple interface provided by the runtime system. These checkpoints can be used to split execution of a long running application by restarting the application from the last checkpoint.

**Resilience and fault tolerance:** Based on the checkpoint-restart mechanism, Charm++ also offers a double in-memory fault tolerance mechanism for applications running on unreliable systems. In this scheme, periodically, two copies of checkpoints are stored: one in the local memory of the PE and the other in the memory of a remote PE. On failure, a new PE replaces the crashed PE, and the most recent checkpoints of chares running on the old PEs are copied to it. Thereafter, every PE rolls back to the last checkpoint, and the application continues to make progress. Other exploratory schemes such as proactive resilience, automatic checkpointing, message-logging also exist in Charm++.

**Malleability and Moldability:** Charm++ jobs have the ability to shrink or expand by invoking a customized load balancer. The main idea is to evacuate chares from nodes which would be removed in case of shrink, and provide them to the new processes in case of expand. The above mentioned benefits of XMAPP languages and Charm++, and their causes, are summarized in Figure 1.10.
Figure 1.10: Summarizing the benefits of XMAPP languages in general, and Charm++ in particular.
Chapter 2

Simple Programs and Basic Chares

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

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. The most important information that these files provide is used to declare some C++ objects as chares. We will learn about it and other contents of interface files in the examples of this chapter.

2.1 Chares: Concurrent C++ Objects

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 chares (objects) on remote processors.
- A chare class may have a category of methods called entry methods, which 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. Throughout this chapter, the examples will use singleton chares, which are a good way of introducing several basic concepts in Charm++. However, most real computational science and engineering applications will be written using chare arrays, signifying collections of chares, which we will describe in the next chapter.
We believe the best way to learn the material of this chapter is to compile the example codes on a real machine and run them. The examples in this chapter (and in the rest of the book) were tested on real machines and are guaranteed to work. An online appendix HERE (but we need url for the print version) provides a guide to Charm++ installation. Please note that the examples in this chapter are very basic and don’t do anything very useful even when they are run in parallel. 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 chare called the mainchare. In interface files, this class is designated by the keyword mainchare. The constructor of the main class is to a Charm++ program what a main() function is to sequential C programs.

Our first program is a Hello world program that introduces many elements common to all Charm++ programs. The interface file for this simple program is shown in Figure 2.1. There is only one class called Origin, which happens to be a chare class designated as a mainchare. The ci file and the description of Origin is required because we must tell the system about it as Origin is a chare class. The Charm++ keyword chare is used to elevate a class to a chare class.

```
1 mainmodule hello {
2   mainchare Origin {
3     entry Origin(CkArgMsg *m);
4   };
5 }
```

Figure 2.1: Hello World: the interface file hello.ci.

Another important thing to note in Figure 2.1 is the module construct—every 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. It 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 ... */ }
```

Returning our attention to the chare class Origin, there is only one method—its constructor. All the constructors of a chare class must be entry methods. 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 Origin method here. All main chares have a standard constructor that takes a pointer to a system defined class (called CkArgMsg) as its only parameter. CkArgMsg is a system class that contains the standard argc and argv as its data members.

The hello.C file is shown in Figure 2.2. Several features in this file deserve explanation. First, notice that the class Origin inherits from CBase_Origin. 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. In particular, for a chare class Foo declared in the ci file, Charm++ generates a new class CBase_Foo from which Foo should inherit. CBase_Origin is such a class. Inheriting from it allows our Origin class to be created as a chare class, and is thus required. The declaration of CBase_Origin (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_Origin (and other classes) are generated in a file called hello.def.h. The names of these files are based on the name of the module; in general these files are named <module name>.decl.h and <module name>.def.h. That is why hello.C includes those two files at its beginning and at its end, respectively. For every module Foo, Charm++ generates Foo.decl.h and Foo.def.h, which contains the declarations and definitions of system classes.

```cpp
#include <iostream>
#include "hello.decl.h"

/** mainchare */
class Origin : public CBase_Origin {

public:
    Origin(CkArgMsg* m) {
        std::cout << "Hello World!" << std::endl;
        CkExit();
    };

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

Figure 2.2: Hello World: the C++ file hello.C.

As mentioned earlier, program execution begins with the constructor of the main chare Origin. This constructor, in this program, includes only two statements. The first prints “Hello World!” to the output stream std::cout. The second statement calls CkExit(), which tells the runtime system to stop the execution of the program on all its processors that are
CHAPTER 2. SIMPLE PROGRAMS AND BASIC CHARES

running this program. See the description of execution and schedulers in section 1.3.1. If exit() is called instead, only the processor that executes this statement will quit without performing the requisite cleanup functions of the runtime system; the remaining processors will hang waiting for something to happen.

2.2 Building, Compiling, and Executing

To compile this program first we need to download and build a copy of Charm++. The Charm++ software can be downloaded as a tarball by following a link on this page: SOFTWARE LINK. Once you download the source code, untar it and change directory to the source code directory. In this directory, you will find a script ./build that can be used to customize the build configuration.

The build script takes the following options:

```
./build <target> <version> <options> [ compiler–options ... ]
```

To build a standard Charm++, set target=charm++. The version will depend on the machine and underlying communication layer you wish to use. On a typical 64-bit Linux machine, you should use version=netlerts-linux-x86_64. On a 64-bit Mac, you should use version=netlerts-darwin-x86_64. For the options, you should add options=smp so Charm++ uses shared-memory mode internally, allowing the Charm++ to share data inside a node using pthreads, instead of launching separate processes inside a node. Finally, you may want to add -g to the end of the command, to enable debugging symbols. Thus, for a 64-bit Linux machine the command you would issue to build Charm++ is:

```
./build charm++ netlerts–linux–x86_64 smp -g
```

Once Charm++ is finished building (assuming there aren’t any errors), a directory will be created that corresponds with your build configuration. In this case, it will be called netlerts-linux-x86_64-smp. Inside that directory, you will find several directories that are useful for building Charm++ programs: bin, include, lib, etc. The Charm++ compiler, charmc can be found inside the bin directory. When writing a Makefile for your test programs, you will want use this path to find the Charm++ compiler.

To compile this program, you first issue a command to translate the interface file:

```
charmc hello.ci
```

charmc is a script provided with Charm++ to can handle such processing automatically. (When you run this command, 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
2.2. BUILDING, COMPILING, AND EXECUTING

the hello.def.h file, which are to be included in the .C file. 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 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). Refer to Figure 2.3 for a graphical display of the compilation process.

![Figure 2.3: Build process of a chare class. If a header file is not used, include the xxx.decl.h file at the top of the .C file in place of the xxx.h file.](image)

At this point, having executed the following commands, you have created an executable file hello:

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

How to execute hello in parallel depends on what machine you are using. The charmrun script, which is created in your folder\(^1\) by charmc when it creates the executable, can be used to run Charm++ programs on most machines. Based on the type of Charm++ installation performed, charmrun is aware of the process required to launch a parallel job.

On a cluster of workstations connected by ethernet, you type the following to run hello using seven processes:

```bash
./charmrun ./hello +p7
```

The +p7 tells the system to use seven processes. 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. On a cluster, charmrun looks for the file nodelist in the current directory or the file .nodelist in the home directory to identify the systems that can be used to launch a job. This file should contain the hostname of the systems that can be used for the execution. We advise you to read the appendix

---

\(^1\)We will use the words “folder” and “directory” interchangeably.

If you want to run this program with seven threads and one process, and you built Charm++ with `options=smp`, you can use `++ppn 7`. The `ppn` option tells the system that you have seven available worker threads per process. Therefore, for a command, `+pN ++ppn K`, where `N == K`, only one process will be created with seven worker threads. In general, `K` must be a multiple of `N`: Charm++ will create `N/K` number of processes, each with `K` worker threads. In SMP mode, a communication thread is created with each process. Thus, in this example, eight total pthreads will be created, where one of the them is dedicated to performing communication operations across processes.

```
./charmrun ./hello +p7 ++ppn 7
```

### 2.3 System Utilities: I/O, Program Arguments

Remember the argument `CkArgMsg *m` for the constructor of the main chare, in the example above? As mentioned earlier, `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` and `++ppn 7` 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`.

Remember in the example program, we used `std::cout` to print to the screen. However, in general we should avoid using the standard print libraries from C and C++ when using Charm++ because printing from multiple nodes may cause ugly interleaving of strings being output to the screen. For ease of use, we have created utilities functions that buffer output and ensure it is safely printed to the screen. As a replacement for `std::cout` we have created `ckout`, which works similarly to iostream, but is safe for parallel output. Similarly, we have created `CkPrintf`, which functionally works the same as `printf`, but is safe for parallel output.

In our example program, you can replace the output line with:

```
ckout << "Hello World!" << endl;
```

### 2.3.1 Example: Hello World with Command Line Arguments

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 9 and 10, which print the
2.4. CREATING MULTIPLE CHARIES

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

class Origin : public CBase_Origin {
public:
  Origin(CkArgMsg *m) {
    cout << "Hello World!" << endl;
    if (m->argc > 1)
      cout << "and Hello " << m->argv[1] << "!!!" << endl;
    CkExit();
  }
};

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

Figure 2.4: Hello World: parsing command line file hello.C.

string argv[1]. Try running this program with an additional argument, which could be your first name:

```
./charmrun ./hello Sanjay +p7 ++ppn 7
```

Hello World!
and Hello Sanjay!!

2.4 Creating Multiple Chares

In order to make use of multiple processors, many chares should be created by the executing program. Presence of multiple chares allows the Charm++ RTS to distribute these chares to different PEs for concurrent execution. In Section 2.1, we mentioned that a single instance of the main chare is created by the RTS, but who creates all other instances of various chares? Since the execution begins with the constructor of the main chare, it is imperative that the main chare creates at least some new chares. These new chares may create other new chares some time during their execution, and so on. It is to be noted that a main chare is also a type of chare, with the special property of being automatically created by the RTS at the program start up.

We now extend the example shown in Section 2.3.1 to show how multiple chares are declared and created. Figure 2.5 presents the interface file for this multichare example with declarations for three types of chares. As before, the Origin class is designated to be a mainchare. The keyword chare is used for declaring a class to be a chare to the Charm++
CHAPTER 2. SIMPLE PROGRAMS AND BASIC CHARES

RTS. Here classes named Interim and Destination are designated to be chares by using this keyword. The constructors of each of these classes are declared to be entry methods since the RTS may need to invoke them remotely for creating instances of these classes. The constructors of chares (except main chares) can accept any set of parameters and can be overloaded in accordance to C++ standards. In Figure 2.5, constructors of both classes, Interim and Destination, take an integer parameter.

```plaintext
mainmodule HelloMultiChares {
    mainchare Origin {
        entry Origin(CkArgMsg *m);
    };
    chare Interim {
        entry Interim(int x);
    };
    chare Destination {
        entry Destination(int x);
    };
};
```

Figure 2.5: Creating multiple chares: the interface file hello.ci.

The C++ code for the multichare example is shown in Figure 2.6. The file begins with inclusion of standard C header (stdio.h) and HelloMultiChares.decl.h, which is generated by charmc by processing the ci file. The HelloMultiChares.decl.h file contains declarations of auxiliary base classes, such as CBase_Origin, CBase_Interim, and CBase_Destination, from which the corresponding chare classes should inherit. It also contains another set of auxiliary classes called the proxy classes. For every chare class Foo declared in the ci file, a proxy class named CProxy_Foo is declared in the .decl.h file. These classes are used to create and access chare classes remotely via their entry methods as discussed later.

Lines 4-18 of Figure 2.6 contains the definition of the chare class Origin. As before, it inherits from the Charm++ defined base class, CBase_Origin. The constructor of this class is executed during creation of a single instance of the class Origin by the Charm++ RTS at the start up. After parsing the input arguments, the class Origin is tasked with creating an instance of the chare class Interim and pass the value of key to it. However, since chares are special C++ objects that may live on different processors than the one that creates them, the standard new call cannot be used. If we call “new Interim(key)”, we will create an instance of the Interim class on the same processor that the main chare is running on. The Charm++ RTS will not be aware of this newly created object, and hence no remote messages can be sent to it. In summary, it would be just a sequential C++ object.

Creation of globally visible chares is one of the tasks in which the proxy class, e.g. CProxy_Interim, associated with the chare class is useful. To create a system-recognized chare,
one should call the \texttt{ckNew} static method member of the system-generated class as is done in Line 16 of Figure 2.6. The arguments to the \texttt{ckNew} call should be same as the arguments that should be passed to the constructor of the chare class.

The \texttt{ckNew} call tells the \texttt{Charm++} RTS to create, at some time in future, an instance of the \texttt{Interim} chare class on some processor of its choice with the parameters specified. Line 21-26 in Figure 2.6 shows the definition of the \texttt{Interim} class. The constructor of the \texttt{Interim} class, when invoked by the RTS as a result of the \texttt{ckNew} call, first prints a notification message and then creates an instance of another chare class, \texttt{Destination}. In this case, it uses the \texttt{ckNew} method of the \texttt{CProxy\_Destination} class to instruct the RTS to create a new chare object.

Line 28-34 presents the definition of the \texttt{Destination} class. Its constructor prints another notification message to the standard output, and calls \texttt{CkExit} to inform the RTS that the user program has completed its tasks. The definition of the auxiliary classes are available in \texttt{HelloMultiChares.def.h} which is included in the C++ file at the end. And, with that, we have a fully functional \texttt{Charm++} program in which multiple chares are created and executed on possibly different processors! We suggest that the reader should try to write this program from scratch, and then compile and run it using the instructions provided in Section 2.2. Although, this is a simple program, running it will help you get in the habit of testing things, and typing the program will reinforce what you are learning.

As you run this program, 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, we 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. But other than that, no ordering can be inferred \textit{even though the constructor of \texttt{Interim} is clearly going to execute after the prints in the \texttt{Origin} constructor}.

Quick summary of main points that were covered in this section:

- Use \texttt{ckNew} method of proxy class to create new chares.
- New chares can be created from any chare, but the main chare must create some chares or generate work to guarantee forward progress.
- Prints from different chares may appear out of order.
- Call \texttt{CkExit()} from the last entry method that the program should execute.

\textsuperscript{2}There are debugging options that make such causally connected prints to appear in order. Refer to the \texttt{+syncprint} command-line option in \texttt{Charm++} manual.
```cpp
#include <stdio.h>
#include "HelloMultiChares.decl.h"

class Origin : public CBase_Orgin {
public:
    Origin(CkArgMsg* m) {
        int key = 1;
        if(m->argc > 1) {
            cout << "<Origin> Hello " << m->argv[1] << "!" << endl;
            if(m->argc > 2) {
                key = atoi(m->argv[2]);
            }
        } else {
            cout << "<Origin> Hello World!" << endl;
        }
        CProxy_Interim::ckNew(key);
    }
};

class Interim : public CBase_Interim {
public:
    Interim(int key) {
        cout << "<Interim> Got the key from Origin" << endl;
        CProxy_Destination::ckNew(key);
    }
};

class Destination : public CBase_Destination {
public:
    Destination(int key) {
        cout << "<Destination> Got the key:" << key << endl;
        CkExit();
    }
};

#include "HelloMultiChares.def.h"
```

Figure 2.6: Creating multiple chares: the C++ file hello.C.
2.5 Chare Proxies and Entry Methods

In the examples so far, we have created chares using the \texttt{ckNew} method which led to execution of their constructors. In a more common scenario, the programmer would like to create chares, get a \textit{handle} or a reference to them, and then remotely invoke various methods on them. Proxies and entry methods are the features provided by Charm++ that enable these capabilities.

As stated earlier, when the \texttt{ci} file is processed by \texttt{charm}, proxy classes are declared and defined in the \texttt{decl.h} and \texttt{def.h} for every chare class. Instances of these proxy classes can be used to remotely address instances of chare classes. There are two ways to obtain an instance of a proxy class that corresponds to the given instance of a chare class: 1) \texttt{ckNew} call returns the proxy to the instance that will eventually be created, and 2) every chare class inherits a member named \texttt{thisProxy} from its \texttt{CBase} class, which is a proxy instance to the given chare. Proxy instances are copyable using the equals (=) operator and can be communicated as parameters to both local and remote methods.

Once a proxy is obtained to a chare, entry methods can be invoked on the chare from anywhere in the executing program using the proxy. Entry methods are specially designated member functions of a chare class that are annotated using keyword \texttt{entry} in the \texttt{ci} file. During the processing of the \texttt{ci} files, for every entry method specified in a chare class’s declaration, a member function is added to the proxy class with the same signature as the chare class. This allows the programmers to call methods on proxy instances as if they were being called on the chares. As was the case with the \texttt{ckNew} call, an entry method invocation using a proxy instance tells the Charm++ RTS to execute, at some time in future, the corresponding member function on the chare.

The interface file for our next example that demonstrates the use of proxies and entry method is shown in Figure 2.7. It consists of a main chare \texttt{Main} with its constructor being the only entry method. The other class \texttt{Compute} is designated as a chare, and hence its constructor is annotated as an entry method. This class is expected to compute area of circles, and thus the value of \texttt{pi} will be passed as an argument to it constructor. The class \texttt{Compute} also has another entry method, \texttt{findArea}, that takes two arguments - a double and a \texttt{bool}. The first argument is expected to the radius of the circle for which the area should be computed, while the second argument tell the \texttt{findArea} method if it should terminate the execution by calling \texttt{CkExit}.

The C++ file for the ongoing example is shown in Figure 2.8. The main thing to notice is on Line 9, where we store the value returned by \texttt{ckNew} in a variable \texttt{sim} of type \texttt{CProxy\_Compute}. This is an instance of the proxy to the created \texttt{Compute} chare. It has the same methods as the chare, but these methods copy the parameters passed to them into a message, put the address of the real chare and the name of the method in its \textit{envelope}, and send it to the processor that hosts the chare. Using the proxy, in Line 10-12, several remote invocation are made on the created chare with different input parameters. The last call (Line 12) passes \texttt{true}
mainmodule Messaging {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };
  chare Compute {
    entry Compute(double y);
    entry void findArea(double radius, bool);
  };
}

Figure 2.7: Method invocation using proxy: the interface file area.ci.

#include <stdio.h>
#include "Messaging.decl.h"

/*@ mainchare */
class Main : public CBase_Main {
public:
  Main(CkArgMsg* m) {
    double pi = 3.1415;
    CProxy_Compute sim = CProxy_Compute::ckNew(pi);
    for (int i = 1; i < 10; i++)
      sim.findArea(i, false);
    sim.findArea(10, true);
  }
};

class Compute : public CBase_Compute {
private:
  double y;
public:
  Compute(double pi) {
    y = pi;
    cout << "Hello from a Compute chare running on " << CkMyPe() << endl;
  }

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

#include "Messaging.def.h"

Figure 2.8: Method invocation using proxy: the C++ file area.C.
as the second argument signifying the end of execution when that entry method is executed.

The implementation of class Compute is simple. In its constructor, the incoming value of \( \pi \) is stored and a hello message is printed. When the method findArea is called, the area of circle is computed using the radius passed as an argument and printed. If the second argument is true, the execution is terminated by calling CkExit.

At this point, we suggest that the reader should write this simple program and run it to get familiar with the idea of proxies and entry methods.

### 2.5.1 Ordering of Entry Method Execution

The program described in the previous section appears to be correct. In fact, on most machines today, it will run correctly. Nonetheless, the program is incorrect! If we run it multiple times, it will run correctly in most cases, but once in a while it may terminate without executing all the invocations of the findArea method. Do you know why?

Charm++ does not guarantee that two invocations by chare A on chare B will be delivered (and executed) in the same order as they were performed. So, what we think of as the last invocation (findArea(10,1)) in the example above may execute before an earlier invocation (e.g. findArea(8,0)), and thus terminate the program prematurely.

There are two reasons because of which Charm++ does not guarantee in-order delivery. First, there is additional overhead or cost in guaranteeing in-order delivery. On many machines, the raw messages may be delivered out of order. Thus, to guarantee in-order delivery, someone somewhere has to keep track of message order using sequence numbers on messages and buffer them until the next in-order message arrives. We would rather not pay that cost unless it is necessary. When it is necessary, the programmer can do the buffering and add the sequence numbers. Later, we will see another notation built on top of Charm++ (SDAG in Section 4) that ensures in-order delivery, but we want to keep the base-line Charm++ flexible to allow either usage.

Second, there are situations where we would rather reorder messages. On each processor core, the Charm++ RTS runs a scheduler, which picks a message (which is a method invocation stored in a general envelope), identifies the chare and entry method indicated on the envelope, unpacks the parameters from the message, and invokes the method using these parameters. As a result of this invocation, new method invocations may get enqueued on its 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 (Section ??) - high priority messages to the same object could execute before low-priority messages sent earlier. This prioritized execution of entry methods presents a strong reason for not enforcing in-order execution.

In order to avoid incorrectness in such situations with out-of-order execution, we need to get used to thinking asynchronously to ensure that the program will work irrespective of the order in which messages get delivered. This thinking is useful in parallel programming
in general. 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 hence it is useful to develop
this asynchronous mode of thinking.

Next, we present a simple design trick to ensure correct termination of the example
presented in Figure 2.8, which is worth using in such situations. Here, we assume what we
are only interested in correct termination of the program and we do not care about the order
in which various \texttt{findArea} queries are executed. To achieve our goal, we modify the example
program such that every invocation of \texttt{findArea} makes a remote invocation on the main chare,
\texttt{Main}, to inform it about its successful execution.

Figure 2.9 presents the \texttt{ci} file for the correct version of the ongoing example. Three modifi-
cations have been made: 1) the main chare, \texttt{Main}, has an entry method called \texttt{doneArea}, which
should be called by the instances of \texttt{Compute} class when they execute the \texttt{findArea} method,
2) the constructor of the \texttt{Compute} class has an addition argument of type \texttt{CProxy\_Main} which
is used to pass a proxy to the main chare, \texttt{Main}, to inform it about its successful execution.

Figure 2.9 presents the \texttt{ci} file for the correct version of the ongoing example. Three modifi-
cations have been made: 1) the main chare, \texttt{Main}, has an entry method called \texttt{doneArea}, which
should be called by the instances of \texttt{Compute} class when they execute the \texttt{findArea} method,
2) the constructor of the \texttt{Compute} class has an addition argument of type \texttt{CProxy\_Main} which
is used to pass a proxy to the main chare, \texttt{Main}, to inform it about its successful execution.

The corresponding \texttt{C++} file is shown in Figure 2.10. The main chare now has an object-
level data member \texttt{count}, which it initializes to ten - the number of \texttt{findArea} invocations it
performs. Notice the constructor call in which a proxy to the current chare (main chare) is
passed using \texttt{thisProxy} in Line 11. Analogous to the \texttt{this} object handle of \texttt{C++} classes, every
chare has \texttt{thisProxy} that can be used to access the proxy to the currently executing chare.
The constructor of the \texttt{Compute} chare class stores the proxy and the value of \texttt{pi} passed to it
(Line 29-30). The entry method \texttt{findArea} on being executed print the area and invokes entry
method \texttt{doneArea} of \texttt{Main} using this stored proxy.

The \texttt{doneArea} method in class \texttt{Main} (Line 17-20) exits once it has been invoked ten times.
Notice that even if the ten invocations arrive out of order, the program will exit only after all
the ten invocations have executed. Also notice that we should not assume that the \texttt{i^{th}} invocation
of \texttt{doneArea} will be for radius \texttt{i}, although the for loop in \texttt{Main::Main} might make us think
2.5. CHAR PROXIES AND ENTRY METHODS

```c++
#include <stdio.h>
#include "Messaging.decl.h"

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

    void doneArea() {
        count--;
        if (count == 0) CkExit();
    }
};

class Compute : public CBase_Compute {
private:
    double y;
    CProxy_Main mainProxy;
public:
    Compute(double pi, CProxy_Main master) {
        y = pi;
        mainProxy = master;
        cout << "Hello from a Compute chare running on " << CkMyPe() << endl;
    }

    void findArea(double r) {
        cout << "Area of a circle of radius " << r << " is " << y*r*r << endl;
        mainProxy.doneArea();
    }
};

#include "Messaging.def.h"
```

Figure 2.10: Handling out-of-order execution: the C++ file areaCorrect.C.
that. The messages carrying the queries and results can, in principle, take varying amounts of time and thus return in a different order.

A chare class, just like other C++ classes, may include regular methods that are not entry methods. These methods will typically be private, called by other methods of the same chare (because, from any other chare, one cannot be sure that this chare is on the same processor. Only entry methods can be called in such a situation.). Such private methods, when called, executed immediately, since they are just regular C++ calls. If foo() is an entry methods, and bar() and barbar() are regular private methods of the same chare, and if foo() calls bar(), then barbar() and then executes some code fragment $C$, these three actions will happen one after other, as expected in regular C++ programs. You can also invoke an entry method as a regular private method(), in which case, it gets called directly. If you invoked it via a proxy (e.g. thisProxy.foo()), though, the method invocation gets enqueued in the schedulers queue, to be invoked at some time in the future.

IS THE ABOVE TOO CONFUSING? Should we spare them this explanation for now?

A quick summary of main points that were covered in this section:

- Only member functions designated as entry in ci file can be invoked remotely.
- Proxy to the chares should be used to invoke entry methods.
- Method invocation on a proxy is a non-blocking operation that tells the Charm++ RTS to execute the corresponding method on the chare at some time in future.
- Entry method invocations may be executed out-of-order and thus the user program should not rely on in-order execution.
- calls to a chare object’s own private methods execute directly, without returning control to the Charm++ scheduler; similarly, direct calls to entry methods that are not invoked on a proxy object are executed directly.

2.6 Complete Example: Concurrent File Search

We now present a complete example program that concurrently searches through a binary input file to find the minimum value. Using the concurrent constructs we have learned, we can write this program in a divide-and-conquer style, using a tree of chares to divide the work and perform a concurrent search. The tree of searching chares will also provide a structure to perform a manual reduction to find the minimum value.

In this example the Main chare will start the computation by creating a Search chare. Each Search chare in this example will take a range of numbers to search in the file. If the input range is larger than the block size, it will create two new chares responsible for
2.6. COMPLETE EXAMPLE: CONCURRENT FILE SEARCH

Figure 2.11: Recursive chare file search: the interface file search.ci.

```c
mainmodule search {
  mainchare Main {
    entry Main(CkArgMsg*);
  }

  chare Search {
    entry Search(bool isRoot, unsigned int lo, unsigned int hi,
               std::string file, CProxy_Search parent);
    entry void result(unsigned int line, double min);
  }
}
```

searching half the range of the parent. Otherwise, the Search chare will perform the search itself.

The format of the input file we are searching is binary: the first four bytes contain the number of doubles, \( n \), stored as a 32-bit integer. Following this, there will be \( 8n \) bytes that store \( n \) double precision floating point values. The objective of this program is to read through all \( 8n \) values and find the lowest one.

The Charm++ interface file is shown in Figure 2.11. The Search chare constructor takes a boolean that indicates whether it is the root or not. Only the first Search chare will be marked as the root, which will cause it to print the minimum instead of sending its value to its parent. The constructor also takes \( lo \) and \( hi \), which stores the range for which this chare is responsible for searching. Next, it takes the filename as a string, so it can open the file to search or send the filename to its children. Finally, it takes a proxy to a Search chare, which is its parent if the chare is not the root.

The initial Search chare that is created from the Main chare is responsible for the entire range (0 to lines_count, shown in Figure 2.12. Each time the constructor of a Search chare runs, it checks if the range (hi–lo) is greater than the defined BLOCK_SIZE. If this is true, the Search chare creates two new Search chares with the range split in half and then waits for their response. This happens recursively until we reach the BLOCK_SIZE. At this point, the bottommost Search chares perform the search and send their values to their parent, shown in Figure 2.13. The parent calculates the minimum of its two children and continues propagation up the tree of chares. When the minimum is finally calculated by the root Search chare, it prints the minimum value found.

In the example, we have defined the BLOCK_SIZE to be 1000. The value of BLOCK_SIZE determines the amount of overdecomposition that we employ. There is a tradeoff between the flexibility gained by overdecomposition and the expense of managing small grain sizes. As we decrease the BLOCK_SIZE we enable the runtime system to exercise more flexibility in
```cpp
#include "search.decl.h"

#include <string>
#include <limits>
#include <stdio.h>

#define BLOCK_SIZE 1000

struct Main : public CBase_Main {
    Main(CkArgMsg* msg) {
        if (msg->argc != 2) {
            CkPrintf("%s: expecting one argument <filename>
\n", msg->argv[0]);
            CkAbort(""");
        }

        char* filename = msg->argv[1];
        FILE* file = fopen(filename, "r");
        if (!file) CkAbort("could not open file\n");

        unsigned lines_count = 0;
        int ret = fread(&lines_count, sizeof(unsigned), 1, file);
        fclose(file);

        if (ret != 1) {
            CkAbort("could not read first line properly\n");
        } else {
            CkPrintf("Read file, %u lines\n", lines_count);
        }

        // search range = [0, lines_count)
        CProxy_Search::ckNew(true, 0, lines_count, std::string(filename), CProxy_Search());
    }
};
```

Figure 2.12: Recursive chare file search: the C++ file (Part 1: Main chare) search.cpp.


```c++
struct Search : public CBase_Search {
  bool isRoot; CProxy_Search parent; int counter;
  unsigned int minLine; double minVal;

  Search(bool isRoot, unsigned int lo, unsigned int hi, std::string filename, CProxy_Search parent) :
    parent(parent), isRoot(isRoot), counter(0), minLine(0),
    minVal(std::numeric_limits<double>::max()) {
    CkPrintf("Search chare: [%d,%d]\n", hi, lo);
    if (hi - lo <= BLOCKSIZE) {
      FILE* file = fopen(filename.c_str(), "r");
      if (!file) CkAbort("could not open file\n");
      if (fseek(file, 4 + lo * 8, SEEK_SET) != 0) CkAbort("fseek failed\n");
      double* val = new double[hi - lo];
      fread(val, sizeof(double), hi - lo, file);
      fclose(file);
      for (int i = 0; i < hi - lo; i++) {
        if (val[i] < minVal) {
          minLine = i + lo;
          minVal = val[i];
        }
      }
      counter++; 
      thisProxy.result(minLine, minVal);
      delete [] val;
    } else {
      const unsigned int mid = (hi - lo)/2 + lo;
      CProxy_Search::ckNew(false, lo, mid, filename, thisProxy);
      CProxy_Search::ckNew(false, mid, hi, filename, thisProxy);
    }
  }

  void result(unsigned int line, double min) {
    counter++;
    if (min < minVal) {
      minLine = line;
      minVal = min;
    } if (counter == 2) {
      if (isRoot) {
        CkPrintf("Found minimum value %f on line %d\n", minVal, minLine);
        CkExit();
      } else {
        parent.result(minLine, minVal);
      }
    }
}
```

Figure 2.13: Recursive chare file search: the C++ file (Part 2: Search chare) `search.cpp`.
migrating chunks of work to achieve a good load balance, which is especially useful when the problem domain is irregular or other non-uniformities arise (e.g. heterogeneous architectures). However, if the BLOCK_SIZE is too small the overhead of managing small chares will decrease performance. In this case, if the BLOCK_SIZE is too small we will incur file system overheads because each chare is opening a file descriptor and reading the file concurrently with other chares. In the following section, we elaborate on the tradeoffs of grain size and discuss quantitative ways of analyzing grain size choices.

2.7 Overdecomposition and Grainsize

We touched upon the issue of controlling grain size in the examples in this chapter. Let us now elaborate on the rationale on how to choose a grain size, and provide some quantitative reasoning about this issue. Programmers who are knowledgeable about parallel programming, but are new to Charm++, often have the following question: all the benefits of Charm++ (that we discussed in Chapter 1) are fine, but doesn’t over-decomposition come with a high overhead? Wouldn’t that make Charm++ impractical, at least for some applications? Even if you don’t have that question, you still are left with a practical question: how fine, or how coarse, should I decompose my data and work?

Let us make our question more concrete. Let the grain size be defined as the computation time an entry method takes, when invoked, before it returns control back to scheduler. That, of course, is just one instance of an invocation on one chare object. Since the program contains a large number of objects, each containing many entry methods, which are being repeatedly invoked, in reality we have a large collection of grain sizes. So we can speak of average grain size, minimum grain size, maximum grain size etc. What should a programmer aim to achieve with their decomposition scheme in terms of grain sizes?

To analyze this question, let us first assume that all the grains are of an equal size, say \( g \) time-units, and the overhead associated with each grain is \( t_o \) time units. This overhead includes scheduling (queuing/de-queuing), and messaging costs. For most current machines, one can approximate this to be a few microseconds (say between one and 5 s). With these assumptions, the execution time on one processor \( T_1 \) becomes \( T + t_o \cdot \frac{T}{g} \), where \( T \) is the sequential execution time of the computation being decomposed. If one then runs this program on \( P \) processors, assuming perfect speedup, the execution time becomes:

\[
T_P = \max \{ g, T_1/P \}
\]

\[
T_P = \max \{ g, \frac{T(1+t_o)}{P+1} \}
\]

The max operator arises here because one cannot execute the program any faster than grainsize of the largest grain. By definition, the computation cannot be decomposed any further. This is an intuitive corollary of Amdahl’s law, which states that the speedup of a computation is limited to \( \frac{100}{k} \), if \( k \) is the percentage of time spent in the sequential part of the computation.
The curves for both $T_1$ and $T_p$ are schematically plotted in Figure 2.14. At the left end, as the grainsize becomes smaller, the overhead dominates and the execution time goes to infinity. At the right end, for $P$ processors, there is not enough parallelism, and eventually the execution time becomes the same as that on one processor. This corresponds to having a single grain that includes the whole computation – hence, no parallelism.

But the interesting point is that there is a long region in the middle which is relatively flat. Looking at the single-processor plot, once the total overhead is brought down to, say, 5%, there is not much benefit to increasing the grainsize further. On the other hand, smaller grainsizes allow one to scale to a larger number of processors, and also gives the load balancers a better chance to equalize the load. So, one should set the average grainsize sufficient large as to amortize the overhead, but no larger. If 5% overhead is tolerable, one can set it to about 20 times more than $t_o$. If the overhead per grain $t_o$ were to be 3 $\mu$s, that’s about 60 $\mu$s!

That was with a uniform grainsize (i.e. all pieces were equal). If they are not equal, what can we say about the desired grainsize distribution? The “max” operator in the equation for $T_p$ suggests that we do not want $g$ to be larger than $\frac{T_1}{P}$. Trying to be somewhat nice to our load balancers by creating not too large work-pieces, we may want to set the upper bound on the grainsize at, say, $100 \cdot t_o$.

Combining the two, we get that we should try to keep the grainsize between 10 and 100 times the overhead, which can be very fine grained indeed. In fact there are other arguments for why the grainsize may have to be larger. For example, in some computations, the memory overhead or the messaging overhead of overdecomposition is very high. The time spent by
the load balancer in collecting statistics and making a decision is also higher with finer grain.

For the current data, let us look at results form a recent experiment. The computational problem being studied involves modeling a 3-dimensional space. Imagine a room in which some places on its boundary (some area along the walls, ceilings, or floor) are held at a fixed temperature. The computational problem is to find the distribution of temperature throughout the room. This problem (described in somewhat simplified manner to avoid technicalities), can be solved using a “relaxation” algorithm: represent the room with a 3-dimensional array, with each element in the array representing the temperature at the corresponding point in space. Assign random temperatures to each point, except at the fixed boundaries which are input to the problem. Use two arrays, one to hold the old values of temperatures and one new. Calculate the new value of each point as the average of itself and its six neighbors along the 3 cardinal directions. Swap the new and old arrays, so what was newly calculated becomes the “old”. And repeat. Repeat until no single point’s temperature changes by much; use some pre-set threshold, e.g., no change is more than 1% or 0.1 degrees.

Let us say we have decided to decompose this problem into smaller cubes, i.e., each chare is a sub-cube of the original cube representing the room. To perform the computation described above for the points at its boundary, each cube needs the points at the boundaries of its 6 neighboring cubes, except the cube at the boundary of the room. Assuming that the room is to be represented by a $2048 \times 2048 \times 2048$ array, how large should each chare be? If the chare is considered to contain a cube of size $k \times k \times k$, what should $k$ be? 16? 512? How to think about this?

First point to note is that irrespective of the value we choose for $k$, the Charm++ code for each chare will remain the same. So, one just has to use a parameter $k$ in the code, and then one can experiment with alternative values. The question is, then: does there exist a large range of grainsize which are “reasonable” as we argued earlier?

The results of an experiment we recently did on a Cray machine using 64 cores spread over 2 nodes is shown in Figure 2.15. As one can see, having several hundred chares per core does not increase the overhead much compared with the minima of the curve. Interestingly, when the grainsize is increased so that there is only one chare per core, our analysis based on Figure 2.14 does not expect any penalty. The curve in Figure 2.14 shows increased time for execution only when parallelism is not adequate to occupy all the processors, i.e., when the number of chares per core is less than 1. However, we observe different results in Figure 2.15. This could be because of the communication-computation overlap, and/or it could also be because of better cache performance one gets with the “blocking” effect of smaller cubes. 

So, our conclusion is: the overhead of overdecomposition is not significant for many realistic problems, and in fact it can provide performance benefits because of cache performance, even when the major benefits such as load balancing as not brought to bear upon a problem.
2.8 Parameters to entry methods: Arrays and other types

short example.. say just sending an array and asking the chare do some basic operation on the array of number (sort? sum? eliminate duplicates?)

2.9 Exercises

2.9.1 Calculating Pi by Throwing Darts

Many mathematical computations use the value of $\pi$, but finding the value of $\pi$ is an interesting task by itself. This exercise is to write a simple parallel program to compute the value of $\pi$ based on one of its known usage. The area of a circle of radius 1 unit is $\pi$, whereas the area of a square of size 2 units, within which such a circle is inscribed, is 4 units. So, if we throw random darts inside the square, and assume that 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 $\pi/4$. To virtually throw darts in a computer, we can use a simple random number generator that is provided by standard libraries included with C/C++. Assuming we are simulating throwing of darts in the positive quadrant only, generating $x$ and $y$ coordinates within range 0 to 1.0 determines the dart placement.

A sequential program fragment for this purpose is shown below.
Assume that `numTrials` has to be a large value in order to get a reasonably correct value of $\pi$. Write a Charm++ program that takes the value of `numTrials` as an input value, and creates many chares to conduct these trials and compute the value $\pi$. You may also want to keep the number of chares being created as an input parameter for experimentation purpose. Use `CkWallTimer` to compute and print the amount of total time spent in running this program.

While writing this program (and running) this program, think about the number of trials each chare should conduct so that parallelization is beneficial despite the cost of creating chares and sending messages. Also run this program to check if you end up calculating the value of $\pi$ correctly. Additionally, try providing different values on the command line for the number of chares that should be created, while holding the value of the `numTrials` fixed, e.g. at 100 million.

Another point of consideration for this program is the generation of the random numbers. Each chare needs to generate a unique random seed; otherwise, if two chares have the same initial seeds, they will generate the same sequence of random numbers, which will defeat the purpose of parallel trials. This can be (roughly) accomplished by creating a function that sets the seed of the chares' random number generator that is a function of some serial number associated with the chares. One can have the main chare, for example, pass such a serial number to each chare, which uses it to set its seed. More sophisticated methods exist for random number generation in parallel, but we will not concern ourselves with that in the context of this exercise.

2.9.2 Recursive Creation of Chares to Compute Fibonacci

In this exercise, we suggest use of traditional doubly-recursive algorithm for calculating $n$’th Fibonacci number directly, based on the following definition:

$$
\text{fib}(n) = \text{if } (n < 2) \ n \ \text{else fib}(n-1) + \text{fib}(N-2)
$$

This formulation has an exponential complexity, and is a rather naive way of calculating $n^{th}$ Fibonacci number, especially since an algorithm with logarithmic time complexity exists! But here, the purpose is to practice use of Charm++ rather than exploring interesting parallel algorithms.

For this exercise, we suggest that the reader follow the pattern described in Section 2.6
2.9. EXERCISES

to compute the \( n^{th} \) Fibonacci number. You can have a main chare which inputs the value of \( n \) and begins the computation by creating a chare of another type, say Fib. The Fib chare should check if it is assigned to compute a large Fibonacci number, i.e. if the number is at a higher position than a preselected position threshold in the Fibonacci series. If so, it should recursively fire two child Fib chares that will compute smaller Fibonacci numbers whose sum is the Fibonacci number to be computed by this chare. If the assigned work to a Fib chare is below the threshold, it should compute the Fibonacci directly.

Note that if you 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 would be overwhelming. The amount of computation per parallel operation, which is creation of Fibonacci chares, and processing of each response from a child chare, will be too small. So, instead you should do some explicit grainsize control: if \( n \) is below some predefined \texttt{THRESHOLD}, calculate \texttt{fib(n)} using a sequential version of the algorithm.

2.9.3 Testing a set of numbers for primality

Write a program based on the outline below. (Note that the program has a few artificial restrictions/elements that are meant to teach you specific concepts. So, please follow the instructions to the letter.)

The main chare generates \( K \) random integers, and fires a checkPrimality chare for each. The chare checks if the number given to it is a prime using a variant of the function below, and returns the result to the main chare. The main chare maintains an array of pairs: \{number, Boolean\}, and prints it at the end. An entry should be added to this array (with the number being tested, and a default value such as “False”) as soon as the chare is fired. In particular, you are not allowed to delay adding the entry after the result is returned by the chare. Make sure that your program does not search the array when a response comes. So, figure out a bookkeeping scheme to avoid it.

Obtain \( K \) from a command line argument. You may use \texttt{rand()} from the math library for generating random integers.

For testing primality, use the following function. For extra credit, modify it so that it is not testing for every \( i \), but (a) avoids testing even numbers except 2 and (b) dont let the loop run all the way to number-1).

```c
int isPrime(const long number)
{
  if(number<=1) return 0;
  for(int i=2; i<number; i++)
  {
    if(0 == number%i)
      return 0;
  }
  return 1;
}
```
Part B (grainsize control):

Measuring performance and improving it via grainsize control:

Grainsize control is a way to improve performance of the above program. Use information from the Charm++ manual about how to pass arrays of data to entry methods, and send a bunch (M) of numbers to be tested to each new Chare, and experiment with different values of M to get good performance. You may wish to read M as a command line parameter, for ease of experimentation. Measure performance by adding two calls to CkTimer() in the main chare, one just before starting creation of checkPrimality chares, and the other after all the results have been returned (but before they are printed), and printing the difference between the timers. You may omit (and probably should omit) printing primality results for performance runs. Vary M and report smallest G for which performance was within 5\% infinite grainsize (i.e. $G = K$). Again, make sure our artificial restriction is obeyed: do not send back the numbers the number being tested (because you are not allowed to search for it anyway).

Part C: Let the numbers being tested be 64 bit random numbers. For simplicity, generate them by concatenating 2 32 bit random numbers.

2.9.4 N Queens

Consider the problem of placing N chessboard queens on an NxN chessboard such that no two of them attack each other; A queen is said to attack another if they are both in the same row, column or diagonal. Given that each row must contain exactly one queen in any solution, a way of representing a partially filled board is to use an array `queens` of size N such that `queens[i]` represents the column number in i’th row which has a queen. A row that is not yet filled can be represented by a special value (such as -1). Write a program to search for all solutions to the N queens problem. Since there are a huge number of solutions, you are to print only the count of solutions. Each `Search` chare is given a board in which first k rows have been filled with non-attacking queens (the `queens` array along with k may be passed as constructor arguments to it). The job of the chare is to fire up to N other chares with new boards, in which row k+1 has been filled with a queen such that it does not attack any already placed queens. Of course, if no placement is possible it should send 0, for the count of solutions, to its parent chare. Otherwise, it should wait for count of solutions from each of the (children) chares it fired, and then return the sum to its parent. The top level chare should print the result.

2.9.5 Sudoku??
In Chapter 2, chare classes and individual chares were introduced. In most applications, creating each of the chare objects one-by-one will be cumbersome to the programmer. More importantly, it is often desirable to collect a subset of the application’s chares into sets or collections, e.g., for the purpose of performing certain operations on all elements of the collection, or to be able to address each chare with a meaningful logical name. Additionally, the collection’s indexing scheme can be used to easily code patterns of communication between chare objects while writing applications. For example, a physical simulation with two-dimensional space can be organized as a collection of chares such that each chare represents a tile of the space being simulated. In this chapter, we will introduce the first, and the most widely used, type of collection in the Charm++ programming model, namely chare arrays.

As the name implies, a chare array is an array of chare objects. A chare array shares some similarities with an array of primitive data type in C/C++. Just as an individual integer in an integer array can be identified by using the bracket operator (“[]”) on the integer array, the paranthesis operator (“()”) can be used on the chare array’s proxy to identify an individual chare in the array of chare objects. In addition, they can be indexed in one through six dimensions using integers. Alternatively, the collections can be indexed using bit vectors or strings. Chare arrays can also be either dense or sparse. For example, a 1-dimensional sparse chare array may include 10,000 chares with indices ranging over 10 million to 20 million. The individual elements of the chare array are distributed across all of the available processing elements (PEs) available to the application. Which chare is placed on which processor is typically decided by the runtime system, although the programmer can also influence or override these placement.

Chare arrays have other interesting characteristics that set them apart from arrays in C/C++. Since they are collections of chares, 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. 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 too deeply into all of the characteristics and features of chare arrays, let us
try to make the idea more concrete in the reader’s mind via examples. We will begin with a
ring example that has a single chare array, consisting of a bunch of chares with communication
arranged in a ring-like pattern, which is traversed multiple times before exit is called.

### 3.1 A Single Ring

This example, Single Ring, illustrates how to create and use chare arrays. Single Ring begins
with creation of a single chare array, called Ring. Consecutive array elements are considered
connected with the last array element wrapping around to connect with the first, thus creating
a ring.

Figure 3.1 shows the interface (.ci) file for the Single Ring program. Similar to the examples
discussed in Chapter 2, it has a mainmodule that contains a mainchare class. Remember,
a mainchare is a chare of which a single instance is created automatically by the runtime
system when the application is started. Next, the interface file declares a chare array class
Ring. The “[1D]” indicates that the array will be indexed as a single dimensional array where
the indices are integers. Other than being declared as a 1D array, the Ring chare array class
is declared in the same way that a chare class is declared. Constructors and entry methods
for chare arrays are declared exactly the same way they are declared for single chare objects
as shown in declaration of Ring.
3.1. A SINGLE RING

Figure 3.2 contains the C++ code for the mainchare class. The mainchare constructor obtains two command line arguments using msg of type CkArgMsg passed to it by the runtime. These parameters are the size of ring to be created and the number of times the ring should be traversed (#trips). It then creates a new chare array of type Ring using the ckNew function in a manner similar to the one used for creating simple chares. The only difference is the requirement to pass the length of the chare array as the last argument. Here, the first two arguments to ckNew are for the constructor of chare array class Ring. Having told the runtime system to create a chare array, the constructor invokes an entry method on a random element to begin work.

```c
class Main : public CBase_Main {

public:

  Main(CkMigrateMessage *msg) {}
  Main(CkArgMsg* msg) {
    int ringSize = atoi(msg->argv[1]);
    int tripCount = atoi(msg->argv[2]);
    delete msg; // Done with message

    // 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 and start at a random element
    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 3.2: Single Ring: main chare class in the C file arrayRing.C

At the time of chare array creation, the Charm++ runtime system assigns the elements of the chare array to the processors using a block cyclic mapping. Many more mapping schemes are available in Charm++, but block cyclic mapping is the default scheme used for any chare array. In the block cyclic mapping, the first $k$ elements are mapped to the first processor, the next $k$ elements are mapped to the second processor, and so on. Here $k$ is a hardwired value selected by the Charm++ implementors. If there are more chare array elements than $k$ times the number of processors, then when the final processor is reached, the runtime system begins with first processors once again and continues through until the last.

Even though the Charm++ programming model does not guarantee that messages are
received in same order in which they are sent, 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 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 3.3 contains the code for the Ring chare array class. The constructor is simple: it
stores the parameters it is passed into member variables for later use. The nextI() returns the
index of the next array element in the ring. When the Main::ringFinished() is called, CkExit()
is called causing the program to terminate.

```c
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 3.3: Single Ring: ring chare class in the C file arrayRing.C

The Ring::doSomething() function is a bit more involved. It starts by displaying some text
to the user. It then determines what to do next. If there are elements left in this traversal of
3.1. A SINGLE RING

the ring, it invokes `doSomething` on the next array element with a decremented counter. Otherwise, if there are no elements left in this traversal and there are more traversals remaining, it starts the next traversal by resetting the `elementsLeft` counter and decrementing the `tripsLeft` counter. Finally, if none of the above conditions are true, it invokes the `ringFinished()` entry method on the main chare object indicating that the ring has completed all of its traversals.

Figure 3.4 presents a graphical view of the control flow in the given example. In the beginning, the main chare instructs one of the elements to begin a ring traversal by invoking the `doSomething()` entry method on the chosen element `i` ((1:begin) in Figure 3.4). The `doSomething()` entry method prints a message and instructs the next element in the array to do the same thing by invoking `doSomething()` on it. A traversal of the ring is complete after each element in the array has executed `doSomething()`, following which the next traversal is started ((2:begin next traversal)). Once all of the traversals have been completed, an entry method on the main chare is invoked causing the program to exit ((3:end) in Figure 3.4).

Figure 3.4: Control flow of Single Ring example program. There are `N` array elements in the ring, and index `i` is the element chosen by the mainchare to begin the ring traversals.

Figure 3.5 contains an example execution of the Simple Ring program. Once again, the `charmrun` command is used to launch the Charm++ program on multiple processors. `arrayRing` is the name of the Charm++ executable. The `+p2` indicates that two processors should be used to run the program. The remaining command line parameters are passed to the Charm++ program, in this case “5” and “3” (see `Main::Main()` in Figure 3.2). The “5” indicates that
there should be five array elements in the ring, while “3” indicates that there should be three full ring traversals before the program exits.

```bash
$ ./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 3.5: Example output of Single Ring program

The output of the program in Figure 3.5 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 the number of full ring traversals (including the current traversal) that 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.

### 3.2 Multiple Rings

In this example, we will explore a slightly more complicated version of the Single Ring example from Section 3.1. This example shows how a program can have multiple sets of chares 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 stopping the program from having completely different computations going
3.2. MULTIPLE RINGS

on in each set of chares.

The major differences between this Multiple Rings example and the Single Ring example 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 moving from one element to the next consecutive element. As a result, even though each ring will be performing similar computations (and in particular, is specified using the same code), the example will show that each computation is independent from the others. Third, to demonstrate modularity, the interface files and source code files for the chare classes will be divided into multiple files.

Figure 3.6 illustrates the program flow of Multiple Rings. The basic difference from the Single Ring example in the control flow is that the mainchare creates several rings, instead of just one. The views for individual rings are very similar to the Single Ring program. Since there are multiple rings, the program will only exit after all of the rings have completed all of their traversals.

Figure 3.7 contains the interface file for the Main chare class. This interface file is fairly similar to the interface file for Single Ring with one notable exception – the `extern` module command on line 10. The `extern` module indicates that there is another module (in another interface file) called `multiRing_Ring` that should be included by this module. The contents of the `multiRing_Ring` module are declared in a different interface file (see Figure 3.8). 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 developed independently of one another.

Figure 3.9 contains some of the source code for the Main class. Remaining code, which includes 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 of each ring that should be completed (`tripCount[i]`). Once the command line has been processed by the constructor of the mainchare, 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. The other entry method, `ringFinished`, counts the number of time it is invoked. When the count reaches the number of rings that were created, it calls `CkExit()` that ends the execution.

Figure 3.10 contains the code for the Ring chare class. This code differs from the Single Ring example as it skips elements while traversing a ring (using `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 further along in the ring traversal. We have used skipping of elements as a simple way to represent non-uniformity
Figure 3.6: Control flow of Multiple Ring
3.2. MULTIPLE RINGS

```plaintext
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 3.7: Multi Rings program: mainchare class interface file

```plaintext
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 3.8: Multi Rings program: ring chare array class interface file
CHAPTER 3. CHARE ARRAYS: INDEXED COLLECTIONS OF CHARES

Figure 3.9: Multi Ring program: mainchare class in the C file

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

public:
  Main(CkMigrateMessage *msg) {}
  Main(CkArgMsg* msg) {
    int *ringSize = NULL, *tripCount = NULL;
    processCommandLine(msg, &ringSize, &tripCount);
    CkPrintf("\n"Array Ring (Multi)\" Program\n\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(); }
  }
};
```
3.2. **MULTIPLE RINGS**

of work among different modules.

```c
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 3.10: Multi Ring program: ring chare class in the C file

Figure 3.11 contains the output from a single execution of this example program. One may find the ordering of the output to be quite confusing. For example, the header that is printed by the mainchare (before any chare array are even created) is displayed halfway through the output after some array elements have already displayed their messages from doSomething method. The cause of this phenomenon is related to the fact that the output of the CkPrintf() function calls needs to be packed into a message sent back to the location where the output is being displayed to the user. These messages may arrive out-of-order and thus the program may appear to have executed out-of-order. As the programs get more
complex and have more and more *asynchronous events* 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.

```bash
$ ./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 3.11: Example output of Multi Ring program

Figure 3.12 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_α[β](δ)” 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.
3.2. MULTIPLE RINGS

This example helps demonstrate one of the advantages of Charm++ over other programming models where the receiving processor needs to expect the incoming message (such as send and receive calls in MPI). For example, writing a program with this type of behavior, where the sending processor may send a message to any receiving processors, is harder to program using MPI since the receiving processor needs to call recv() before the message arrives. However, it is not predetermined which processor is going to receive the message. In this example, the destination processor is chosen at random, but the idea is easily extended to any situation where one or more processors may or may not receive a message depending on some condition on the sender’s side.

$ ./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[3](0): tripsLeft = 2, from [1](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](0)
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 3.12: Modified example output of Multi Ring program

Another strength of the Charm++ programming 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 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.

3.3 Reductions

As the name implies, a reduction is an operation that reduces a set of values into a smaller set of values (typically, many values reduced to a single value) according to a specified operation. For example, to sum an array of integers we would repeatedly apply the addition operation to the values within the array until we processed the entire array and arrived at a single final value. One of the requirements for a reduction is that the operation being applied has to be both commutative\(^1\) and associative\(^2\). This allows the operation to be applied to the individual values within the original set in any order.

The function `reduceArray` in Figure 3.13 is an sequential example of the typical way an array of integers can be summed. In the example on line 4, an intermediate value \(r\) is created and initialized to the identity value (0) 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 (integer addition) is both commutative and associative, we can modify the code of `reduceArray` 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 (see `reduceArraySplit` (lines 12-24) in Figure 3.13). In a sequential program, splitting the array is not beneficial, but it demonstrates how commutativity and associativity allows the operations to be reordered, which is helpful in the parallel

---

\(^1\)An operation \(\odot\) is said to be commutative if \(A \odot B = B \odot A\). Integer multiplication is an example of a commutative operation (\(2 \times 3 = 3 \times 2\)). Integer division is an example of an operation which is not commutative (\(2 \div 1 \neq 1 \div 2\)). Note that floating point operations may not be truly commutative/associative.

\(^2\)The operation \(\odot\) is said to be associative if \((A \odot B) \odot C = A \odot (B \odot C)\). Integer addition is an example of an associative operation (\(2 + 3 = 3 + 2\)). Integer subtraction is an example of an operation which is not associative ((\(5 - 3\) - 2 \(\neq 5 - (3 - 2)\)).

\(^3\)An identity value, \(I\), of an operation, \(\odot\), is a value such that \(a \odot I = a\). For example, zero is the identity value of addition (\(a + 0 = a\)), and one is the identity value of multiplication (\(a \times 1 = a\)).
3.3. REDUCTIONS

```c
// Perform a reduction on an integer array
// r = a[0] + a[1] + ... + a[len-1]
int reduceArray(int *a, int len) {
    int r = 0;
    for (int i = 0; i < len; i++)
        r += a[i];
    return r;
}

// Perform a reduction on an integer array by splitting into two halves
// r = (a[0] + ... + a[len / 2]) + (a[len / 2 + 1] + ... a[len - 1])
int reduceArraySplit(int *a, int len) {
    // add the values in the first half of 'a'
    int r1 = 0;
    for (int i = 0; i < len / 2; i++)
        r1 += a[i];

    // add the values in the second half of 'a'
    int r2 = 0;
    for (int i = len / 2 + 1; i < len; i++)
        r2 += a[i];

    return r1 + r2;
}
```

Figure 3.13: Two versions of a function that performs a serial reduction on an array of integers.
context. Note the two for loops in the reduceArraySplit function are completely independent of each another and thus can be performed in parallel without any synchronization. The results of the two different loops will still need to combined to obtain the final result using the same operation applied to the initial values themselves. In reduceArraySplit, we have only divided the work into two equal halves; however, the reduction computation could be broken down into even smaller parts.

In the next section, we will show how reductions can be performed in parallel on chare arrays: the user specifies the data from each element in the chare array to contribute and this data is efficiently reduced by the runtime system. Charm++ provides built-in functions for computing the sum, product, average, maximum, minimum, etc. for integer and floating point arrays. A full set of all the operations supported can be found in the Charm++ manual.

### 3.3.1 Reductions over Chare Arrays

To perform a reduction over a chare array, each element chare must make a call to the contribute method to deposit their part of reduction. The contribute method is one of the system-defined methods that a chare class Foo inherits from the base case CBase_Foo. Note that in Charm++, reduction are asynchronous and are overlapped with normal execution. Hence, a call to the contribute method does not wait for completion of such calls on other chares; instead, it returns immediately. (i.e. is not a barrier). The call passes the chare’s contribution to the runtime system and then returns to continues execution of the caller. Under the hood, the Charm++ runtime performs the reduction incrementally as values are asynchronously contributed by the members of the chare array. Once all the elements have called contribute, and the reduction computation is finished, a callback is invoked.

A callback is a general way to safely perform some action specified by the creator of the callback at a later time, say, when some operation is complete. A callback may be a simple C++ function to invoke, or a broadcast to a certain chare array for instance. Many different types of callbacks that can be created are explained in detail in the Charm++ manual. A common case is when the callback is directed to a specific entry method of a specific chare, as illustrated below.

A callback is constructed using the CkCallback class:

```cpp
CkCallback cb(...);
```

If an entry method will be used as a target of a callback, it must be marked with the reductiontarget attribute in the .ci file, as shown in figure 3.14a on line 4:

```cpp
entry [reductiontarget] void entryMethod(double reducedValue);
```

Then, to create a callback to an entry method for a reduction, a CkReductionTarget callback can be created:

```cpp
CkCallback cb(CkReductionTarget(ClassName, entryMethod), proxyToClassName);
```
3.3. REDUCTIONS

Note that if the proxy is to an entire chare array, the method will be broadcast to the array.

Figure 3.14 contains a simple example of using a reduction to sum all the indices in a chare array. In the example, the main chare instantiates the Elem chare array on line 11 of Figure 3.14b, passing a proxy back to it through the constructor. Hence, every element of Elem will have a proxy to the main chare.

In the constructor of Elem, the index of the Elem, which will have the range \([0, n]\) where \(n\) is the length of the chare array, is stored in value. A callback is created back to the entry method printResult of the main chare, using the proxy mainProxy passed to the constructor on line 24. The contribute method is then called by each chare array element, depositing its value. The contribute method takes the number of bytes being deposited, a pointer to the data, the reduction operation to use (the CkReduction contains the built-in functions),\(^4\) and the callback to execute when the reduction is finished.

When the reduction is complete, the program will print a number equal to \(\frac{n(n - 1)}{2}\), which is the closed form for calculating the sum from 0 to \(n - 1\).

Although in our example, the computation finishes after the reduction completes, and nothing else is going on when the reduction is progressing, neither of those are essential properties of reduction. After contributing into a reduction, a chare may process entry methods and continue parallel execution as normal. The completion of reduction just injects the callback into this ongoing message-driven execution.

3.3.2 Multiple Reductions

Since reductions are asynchronous in Charm++, multiple reductions over the same chare array or multiple chare arrays can be executed concurrently. To perform multiple reductions over the same chare array correctly, multiple contributions must be called in the same order on each element of the chare array. So, you can make a call to contribute for the first reduction from a chare, and even before the corresponding callback is called, signifying that the contributions from all elements of the chare array have been received and processed, you can contribute into a second reduction. The two reductions will proceed to execute across processors in the background, concurrently with the other activities of the calling chare array. (Of course, sometimes the application logic itself requires one to wait for the results of reduction before proceeding; but the point is the Charm++ system doesn’t force you to wait). Other than this, the semantics are identical to performing a single reduction.

\(^4\)In addition to several built-in reduction types, such as max, min, and sum, user-defined reductions can be defined and passed to contribute. This is explained in detail in the Charm++ manual.
CHAPTER 3. CHARE ARRAYS: INDEXED COLLECTIONS OF CHARES

(a) reduction.ci

```c
mainmodule reduction {
    mainchare Main {
        entry Main(CkArgMsg*);
        entry [reductiontarget] void printResult(int result);
    }

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

(b) reduction.cc

```c
#include "reduction.decl.h"
#include <cstdlib>

#define DEFAULT_NUM_ELEMS 10

class Main : public CBase_Main {
public:
    Main(CkArgMsg* msg) {
        int numElems = msg->argc > 1 ? atoi(msg->argv[1]) : DEFAULT_NUM_ELEMS;
        CkPrintf("reduction: number of elements = \%d\\n", numElems);
        CProxy_ELEM::ckNew(thisProxy, numElems);
    }

    void printResult(int result) {
        CkPrintf("result = \%d\\n", result);
        CkExit();
    }
};

class Elem : public CBase_ELEM {
public:
    Elem(CProxy_Main mainProxy) {
        int value = thisIndex;
        CkCallback cb(CkReductionTarget(Main, printResult), mainProxy);
        contribute(sizeof(int), &value, CkReduction::sum_int, cb);
    }
    Elem(CkMigrateMessage*) { }
};
#include "reduction.def.h"
```

Figure 3.14: Simple example of how to use reductions with chare arrays.
3.4 Prefix Sum with Recursive Doubling

Next, let us consider a well-known problem called the prefix sum. The sequential formulation of the problem is simple enough: given an array \( A \), consisting of numbers, create an array \( B \) such that the \( i \)th element of \( B \) is the sum of all the elements of \( A \) to the left of and including the \( i \)th element of \( A \). A naive algorithm will run a loop for calculating each element of \( B \). However, a moment’s reflection tells us that an element of \( B \) can be calculated from the previous element of \( B \). This leads to the following sequential code:

\[
\text{for } i = 1 \text{ to } N \quad B[i] = B[i-1] + A[i];
\]

// rewrite as proper C++ code

How can we do this in parallel? Can we do this in parallel at all? Given that each element of \( B \) depends on its previous element, it appears that we have a sequential chain of dependencies, thwarting any attempt at executing this in parallel. It turns out that there is a beautiful algorithm called recursive doubling that can accomplish this in parallel at the cost of some extra work. To explain this algorithm, we would first consider a simplified version. Assume that we have a chare array of size \( N \), and each chare holds exactly one number. We would like each chare to compute another number which is the sum of all the numbers originally held by all the chares to its left and itself.

The algorithm proceeds in phases. In each phase each chare communicates with at most one other chare to the right of it. In the first phase, every chare sends its value to the chare to its right, i.e., to a chare at distance 1 from itself to the right. Of course, since the last chare has no chare to its right, it doesn’t send anything. Every chare that receives a number, adds it to its value, as shown in the second row of Figure 3.15. In the second phase, the communication distance doubles: every chare sends to the chare to its right at a distance of two, if one exists. Again, they all add up the numbers they receive. In \( \log P \) phases, the distance is not less than \( N \) any more, so there is no one to send to for any chare, and the algorithm stops. You can verify that the value computed in each chare at the end is indeed the sum of all the original values to its left and itself. E.g., chare 3 has 17, which is the sum of 5, 3, 7, and 2.

The total number of additions in this algorithm is \( P \log P \), instead of just \( P \) additions if it were done sequentially. But at least it can be done in parallel. It will take time proportional to sending of \( \log P \) messages, as each phase requires some chare sending and some chare receiving one (but no more than one) message. The alternatives are much worse: we can send messages in a chain from chare 0 rightwards, carrying the partial sum. But this will take \( P \) messages (sends and receives) in a chain. Collecting all data on one processor has a similar complexity, because of the need to receive \( P \) messages one after the other on chare 0, and may be infeasible if the number of chares is huge (an unlikely problem... but at least we note the larger amount of memory needed on chare 0 to hold all the data).

Now, let’s turn to implementing this algorithm in Charm++. The interface file, prefix.ci is shown in Figure 3.16. There is a main chare Main, and a chare array of type Prefix. Prefix has a step entry method to step through the phases of the algorithm, and a passValue entry
Figure 3.15: At each stage, the distance that each chare’s value is sent doubles.

```
mainmodule prefix {
  readonly CProxy_Main mainProxy;
  readonly int numElements;
  readonly CProxy_Prefix prefixArray;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry [reductiontarget] void done();
  };

  array [1D] Prefix {
    entry Prefix();
    entry void step(int);
    entry void passValue(int stage, int value);
  };
}
```

Figure 3.16: prefixSum.ci.
3.4. Prefix Sum with Recursive Doubling

method, to receive the data from left.

The main Chare (Figure 3.17) creates the prefixArray, and waits for the done entry method to be called via a reduction, upon which it terminates the program. The Prefix constructor simply creates a local random initial value (in real applications, this value will come from the application, as we will see in other examples later), and initiates the first phase of the algorithm with distance 1.

The interesting code is that of the step and passValue methods. In each step, a chare checks if it is past the last stage, by checking if the distance it is supposed to communicate with is not smaller than numElements of the chare array. If so, it prints its final value, and contributes into the reduction going to the main chare. Otherwise, it sends its value to the chare at the current distance away from it to the right, if it exists. Now, it can wait for a message from the left (via passValue) before going to the next stage. But it is possible that there is no chare to its left at the current distance (as, for example, will be the case for chare 0 in the first phase), and if so, it must go to the next stage without waiting for any message, which it does by calling step again, after doubling the distance. Otherwise, the chare returns control to the scheduler, and will be called again, via its passValue method, when a message from the left is available. In passValue, the chare adds the incoming value to its current value, doubles the distance and calls the step to go to the next stage. This seems like a faithful implementation of the recursive doubling algorithm we described above in Charm++. But it is wrong. Take a few minutes to see if you can spot the problem.

5 Alternatively, one could just loop here and send the remaining messages instead of making the recursive call.
#include "prefix.decl.h"
#include <math.h>

/** readonly */ CProxy_Main mainProxy;
/** readonly */ CProxy_Prefix prefixArray;
/** readonly */ int numElements;

class Main : public CBase

public:
    Main(CkArgMsg* msg) {
        mainProxy = thisProxy;
        numElements = 8; //default
        if(msg->argc > 1) numElements = atoi(msg->argv[1]);
        delete msg;
        prefixArray = CProxy_Prefix::ckNew(numElements);
    }
    Main(CkMigrateMessage* msg) {}
    void done() { CkExit(); }
};

class Prefix : public CBase

public:
    int* valueBuf, *flagBuf, value, stage, numStages;
    Prefix() : stage(0) {
        numStages = log2(numElements);
        valueBuf = (int*)malloc(numStages*sizeof(int));
        flagBuf = (int*)malloc(numStages*sizeof(int));
        srand(thisIndex);
        value = rand()%10 + 1;
        //CkPrintf("Chare %d generated value %d\n", thisIndex, value);
        step(value);
    }
    Prefix(CkMigrateMessage*) {};

    void step(int value) {
        if(stage >= numStages) {
            //CkPrintf("Chare %d on PE %d finished with final value %d\n", thisIndex, CkMyPe(), value);
            CkCallback cb(CkReductionTarget(Main, done), mainProxy);
            contribute(sizeof(int), &value, CkReduction::sum_int, cb);
        } else {
            int sendIndex = thisIndex + (1 << stage);
            if(sendIndex < numElements) thisProxy[sendIndex].passValue(stage, value);
            if(flagBuf[stage] == 1) { updateValue(); }
            else if(thisIndex - (1 << stage) < 0) {
                stage++;
                step(value);
            }
        }
    }

    void passValue(int incoming_stage, int incoming_value) {
        flagBuf[incoming_stage] = 1;
        valueBuf[incoming_stage] = incoming_value;
        if(flagBuf[stage] == 1) { updateValue(); }
    }
The problem is that passValue messages can be processed out of order. Consider chare with index 2. It is supposed to get a message from chare 1 in the first stage, and chare 0 in the second stage. But chare 0 is not waiting for any other chares (there are no chares to its left). So, it may send a stage 2 message, after sending its stage 1 message to chare 1) pretty early on, and chare 2 might get it before it gets the first message sent by chare 1. This will mess up the careful summing that the algorithm is doing via recursive doubling and get to a wrong result.

It is tempting to fix this with a reduction with no value (aka a barrier) between stages, followed by a broadcast to commence the next stages. Think for a moment about why that is not a good idea.

A reduction and a broadcast are expensive operations, taking $O(\log P)$ time, and so each of the log $P$ stages will take a much longer time. A better solution is to keep track of such synchronization (here: which actions should be allowed to proceed) based on local data and some extra book-keeping. A value sent to passValue must also be accompanied by a stage number. Since we are explicitly tagging stage numbers, we can calculate the distance from the variable stage and determine if the chare’s value should be updated immediately, or if it needs to wait for a value from an earlier stage to arrive before proceeding.

We implement this via buffers. Two parallel arrays allow us to hold both all incoming values, and the stage they came from. Now when a chare receives a value (in passValue), we first buffer that value and mark (with flagBuf[stage]) that we have a value ready from that stage. Only then does the chare see if it should update its current value. If it can, it does so and proceeds to the next step.

At the start of a new step, the chare first sends out its value as before, but now it also checks to see if it already has the incoming value it needs for the stage it is now on. If it does, it updates its value and can immediately proceed to the next step. If it does not have a value buffered, it checks if the incoming distance for its current stage is greater than its own index. If this is the case, the chare will no longer be the recipient of values, so it can immediately proceed to the next step.

Finally, if neither of those conditions are true (meaning the chare did not have a buffered value and is still waiting on incoming values), control returns to the passValue function, since the chare has nothing to do except wait on incoming values. Once the needed incoming value arrives, the chare once again updates its value and proceeds to the next step.

### 3.5 Multidimensional Chare arrays

Chare arrays can also be multidimensional (from 1-D to 6-D). Creating and using a multidimensional chare array is very similar to a single dimensional chare array. The major difference is that `thisIndex` is a struct that holds multiple index values (one for each dimension). For 2-D arrays, the two dimensions can be accessed using `thisIndex.x` and `thisIndex.y`. For 3-D,
thisIndex.z is added. For indexing beyond 3-D chare arrays, see the Charm++ manual.

To demonstrate multidimensional chare arrays, we will show how to calculate a 5-point stencil in Charm++.

3.5.1 Description of 5-Point Stencil

In a 5-point stencil application, there is a \( n \)-dimensional grid of values that are updated in discrete timesteps. In this example, for simplicity, we will use a 2-D grid. Typically, each point in the grid represents a specific quantity: heat or temperature, for instance. The idea is to simulate how these values change over time according to a specified calculation. During each discrete timestep, every value in the grid is updated using an equation that combines the neighboring values in the grid. For a 5-point stencil, each grid value is updated by averaging a grid point with its four neighboring elements’ values from the previous timestep.

Typically in these simulations, the borders of the grid are held at a fixed value. For example, one or more elements at the center of the 2-D grid could be held 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 the simulation will eventually converge to a stable state. That is, the magnitude of the largest value change for any of the elements will reduce as time goes on. 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 the program can exit.

3.5.2 Parallelizing 5-Point Stencil

To parallelize this calculation, the 2-D grid can be divided in both dimensions to create a 2-D array of equally sized tiles. Each tile will contain an equal number of elements from the original grid. This 2-D array of tiles can be represented as a 2-D chare array, as illustrated by Figure 3.18. On the left is the grid of values (the data points in the simulation). On the right is the 2-D chare array. The two images are overlapped in the center to show how each 2-D chare array element will contain a portion of the grid (i.e. one tile per array element).

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 3.19). Each chare object will have 2-D array of elements that represents its local data. This local data array will have two more elements in each dimension to hold the remote neighboring values. That is, if each tile contains \( n^2 \) elements, then the local data array will be \((n + 2)^2\) in size to store the remote data. This allows for a one element border around each tile which will hold \textit{ghost} information from each of the four neighboring tiles.\(^6\) The term ghost data refers to the border data passed by each

\(^6\)For simplicity, we will calculate the stencil periodically, that is the tiles located on the edge of the overall
tile that will be used as input to the element calculations for the neighboring tile’s border element’s calculations. \textit{Ghost} is used because once the data arrives, it can only be read, but since the elements do not actually reside there they cannot be written.

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. The actual tile data is in the center area of the \((n + 2)^2\) data array. Therefore, 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 determining when the simulation is complete. That is, detecting that no non-fixed element changed by more than the error tolerance. In other words, the maximum value change seen on any tile was less than a specified error tolerance. Parallelizing this process is a straightforward application of a reduction. Remember that a reduction can perform any commutative and associative mathematical operation. In this case, the mathematical operation of \textit{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 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

---

grid will exchange ghost data with tiles on the opposite edge of the grid. As a result, all tile objects will both transmit and receive four ghost messages, even if they are on the edge of the 2-D chare array.
Figure 3.19: Communication pattern of tiles in the 2-D stencil application.
sent to the main chare object. Based on the global maximum value change, the mainchare 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.

### 3.5.3 Code for Parallel 5-Point Stencil

![Diagram](image)

**Figure 3.20:** Logical flow of 2D 5-point stencil execution from the point of view of a single tile object ($\text{tile}[x,y]$).

Figure 3.20 shows the logical flow of the 5-point stencil application from the point of view of a single chare array element. As with all Charm++ applications, the application begins by calling the mainchare’s constructor. For reference, Figure 3.21 contains the relevant portions of the mainchare object’s interface and header files.

The source code for the mainchare class is located in Figure 3.22. The constructor begins by processing the command-line parameters. After that, 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 2-D 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’ by invoking the `Tile::startStep` method.
mainmodule main {
  readonly float targetDiff;
  readonly int gridWidth;
  readonly int gridHeight;
  readonly int tileWidth;
  readonly int tileHeight;

  mainchare Main {
    entry Main(CkArgMsg * msg);
    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 3.21: The interface and header files for the main chare object in the 2-D stencil application.
3.5. MULTIDIMENSIONAL CHARE ARRAYS

```cpp
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(CkReductionTarget(Main, reductionCallback), thisProxy);
    grid.ckSetReductionClient(&cb);

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

Figure 3.22: The main chare object’s constructor (entry-point of application).

```cpp
void Tile::startStep() {
    const int thisX = thisIndex.x;
    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 receives 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 receives 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 receives 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 3.23: The Tile::startStep() method in tile.C.
CHAPTER 3. CHARE ARRAYS: INDEXED COLLECTIONS OF CHARES

Broadcasting the `Tile::startStep` method triggers the beginning of a timestep. When invoked, this method sends ghost messages to each of its four neighbors. The code for the `Tile::startStep` function is located in Figure 3.23.

As a result of all the tiles sending ghost data to each of their neighbors, each tile will also receive four ghost messages, one from each neighbor. There are four functions that handle receiving the incoming ghost messages and copying the incoming ghost data into the associated border elements in the data array (refer to Figure 3.19). Once the incoming ghost data has been copied, each of these functions call the `Tile::countEvent` just like `Tile::startStep` does.

The `Tile::countEvent` method, shown in Figure 3.25, increments a counter and when that counter reaches five, it resets the counter and makes a call to `Tile::doCalc` which does the actual computation on the elements. This is done to make sure the calculation does not start until the tile receives all of the ghost messages. 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
Figure 3.25: The Tile::countEvent method in tile.C.

is important to make sure that all five events occur before a tile proceeds with its own local computation.

Once the counter in the Tile::countEvent function reaches five, Tile::countEvent makes a call to Tile::doCalc, which starts the actual calculation.

The Tile::doCalc function, which is shown in Figure 3.26, updates 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 straightforward. The only complication is the starting value for the inner for loop (the x for loop). The local (0,0) element, actually (1,1) in the data array, of each tile is held at a constant value of MAX_VAL. Because of this, the loops need to skip this element and not update it. Therefore, 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 105-106). Once all of the values have been updated, the Tile chare 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 method swaps the points between 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 actual element values for the tile. If a single buffer were used, there would be a data dependency problem. As the for loops started updating the values of some elements (i.e. if line 104 wrote to tileData instead of scratchData), it would change the input values used in future iterations of the 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

7Notice 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() {
    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
    float* tmp = tileData;
    tileData = scratchData;
    scratchData = tmp;
    enforceConstants();
}

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

Figure 3.26: Calculation function and enforce constants function for the Tile class in tile.c.
3.6 Exercises

In the following exercises, assume that the parameters (usually denoted by upper case letters M, N, K..) are to be obtained from the command line.

1. A simple experiment with reduction: In the single ring example, add a contribute call to do a sum reduction each time the message passes through a chare. The value you contribute should be just the tripsLeft parameter that was passed to you. The callback should be a reduction target method in main chare, which just prints all the added up values from each iteration. Take care to ensure each element contributes exactly once.
$./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 3.28: Output of the 2-D 5-point stencil program run using four processors.

2. K-means Algorithm: Give a collection N points in 2 dimensions, optimally classify them in K clusters using an iterative refinement algorithm, and output only the centroid of each cluster. (for simplicity, we will use 2 dimensions, the real problems involve a larger number of dimensions). More concretely: create a chare array called Points of N chares, each with M/N data points. The constructors initialize each data point with random X and Y coordinates, (0 <= X, Y < 1.0). The main chare generates K random points as an initial guess for centroids of the K clusters, and broadcasts them (as an array of x-y pairs) to the Points chare array, to an entry method called Assign. This method decides, for each point it owns, which centroid it is closest to. It then contributes into 2 reductions: one a sum of points it added to each cluster (so an integer array of size K) and another a sum of X and Y coordinates for each cluster (so, an array of 2K doubles). The target of the reductions are the UpdateCounts and UpdateCoords methods in the main chare. When both reductions are complete the main chare updates the centroids of each cluster (simply calculate, for each of the K clusters, the sum of X coordinates i’th cluster divided by the count of points assigned to i’th cluster, and similarly for Y.) The algorithm then repeats the Assign (via broadcast) and Update steps, until the assignment of points to cluster remains unchanged. We will approximate this by calculating (in the main chare) the changes to any centroid,
and when no centroid coordinate changes beyond a small threshold $T$ (say 0.001), we will assume the algorithm has converged.

**Part B:** Reduce the number of reductions in each iteration from 2 to 1. Hint: there are 2 ways of doing this: first involves approximating the counts to be double precision numbers and the second involves writing a custom reduction).

**Part C:** The “no change to centroids above a threshold” method above is an approximation. Implement a more accurate method by using an additional reduction of the number of points which have changed their “allegiance” (i.e. the cluster to which they belong in each iteration. If this number is 0, the algorithm has converged. Again, as in part B, reduce the number of reductions to just 1 if possible.

3. Implement a distributed hash table using 1D chare arrays. The keys are 64-bit integers, and the data is simple struct (or just another 64-bit integer) Phase 1: main chare creates chare array Table, each element generates keys and (random) data, and stores it. Phase 2: a client array (created by main chare) is triggered by a broadcast to generate a random number of keys, and requests the associated data from the corresponding chare array element of “Table”. When each element receives all the data it requested, it contributes into a reduction. Maybe it should be the same chare array (client and Table). After creating its portion of the table, they all make requests. This will illustrate that (so, the exercise text is: Note that even after contributing into a reduction, a chare can keep responding to others.

4. You will implement a simplified version of distributed hash table for this example. Such tables are needed, for example, when a large collection of key-data pairs must be stored on a parallel computer, in a situation where the entire table is too large to fit on one processor. We will simplify it by assuming the keys range from 0 to just a million (or $M$, in general), and the key as well as data associated with each key is just a single long integer. For this exercise, create a chare array $A$ of $N$ elements with entry-methods request, and response, in addition to its constructor. The constructor will create a table of size $M/N$, where $M$ is the total number of keys. The $i$th chare stores keys starting with $M*i/N$. (So, if $M$ is 10,000 and $N$ is 10, chare 0 stores keys 0..999, chare 1 keys 1,000..1,999 etc.). The constructor populates its share of key-data pairs, by generating random numbers for the “data” for each key.

It then generates $K$ random keys (the set of keys it is interested in) in the range 0..$M$, and sends requests for the appropriate chare array element to fetch the data corresponding to each key it is interested in. It allocates a table $T$ of size $K$ of key-data pairs. Each request includes not only the key, but also the index in $T$ where this key has been kept, and the requestor’s chare array index.

When a chare receives a request for a key, it sends a response to the requestor with the
data as well as the supplied index. When a response arrives, a chare stores it in the right place in $T$. When all the requests a chare has made are fulfilled it does a dummy calculation (here, let us say it will add the data values up), and then contributes the sum into a reduction targeted at a “done” method in mainChare. Note that even after contributing into a reduction, a chare can keep responding to others. The done method simply prints the sum and terminates the program.

5. **Data Balancing:** Assume you have a 1D chare array $A$. Each chare (say $A[i]$) in it holds a vector of numbers. The size of this vector is different on different chares (say $size_i$ on $A[i]$). Your task is to equalize the load on all processors by exchanging the numbers. It is not necessary to do minimal data movement, but it’s desirable. The balance at the end needs to be almost exact. If there are a total of $N$ numbers, and $v$ chares, there should be between floor($N/v$): ceil($N/v$) items on each chare. Note that the only way to send information to another chare is by sending an (entry) method invocation to it.

A. There are many distinct algorithms possible. Sketch the alternatives without coding them, and write cost estimates for them. Keep in mind that the simplest (i.e. approximate) cost model in Charm++: entry methods invocation’s cost $\alpha + n\beta$, where $\alpha$ is a fixed cost, and $\beta$ is a per-byte cost. For the sake of intuition, you may assume $\alpha$ is about a thousand times larger than $\beta$, say a microsecond vs a nanosecond. Reductions and broadcasts of size $N$ data on $P$ processors cost $\alpha \log(P) + N\beta$. Keep in mind that many (but not all) of the algorithms for this problem have two phases: first phase to identify who should send how many numbers to whom, and second to actually do the data exchange. Make sure to write your time estimates for both phases. Compare two of the interesting algorithms in terms of cost, performance tradeoffs if any (e.g. is each algorithm better in different scenarios), scalability and coding complexity. By scalability, here, we mean how well the algorithm behaves with a large number of chares and/or a large number of physical processors.

B. Code one of the algorithms you identified above.