A Tutorial Introduction to Charm++

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

January 3, 2025

Contents

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 and laptops 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 2022 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 gave the parallel computing community a run for its money. They kept making computers faster, year after year, for 30+ years after that (and kept the well-known Moore's law going for 40+ years). The frequencies (and therefore raw single-thread speed, to a large extent) stopped increasing around 2003. And the vaunted Moore's 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 Moore's 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 Moore's law (i.e. by decreasing size of the transistors), the only remaining reasonable way of using them was to 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 system" is loosely defined as the part of the software stack 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.

1.1.1 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 inherently limits how the runtime system can adapt to these changes.

The XMAPP methodology involves decomposing 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 ensuring that there are a sufficiently large number of them to allow the runtime system to rebalance work as necessary. The 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. 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

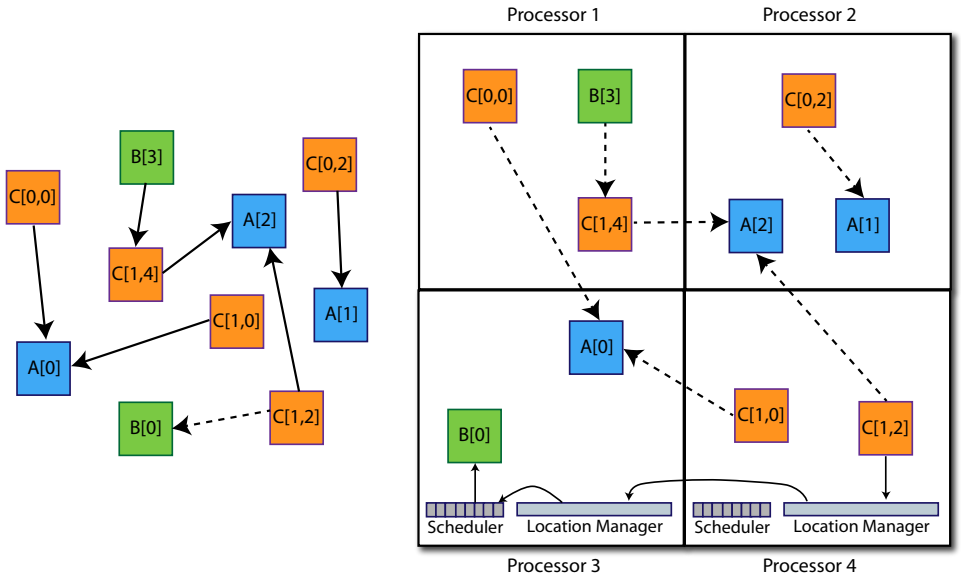

Figure 1.1: Overdecomposition: User's view (left figure) of the program does not involve processors. The runtime system maps the work units to processors and schedules their execution (right figure).

a message to another work unit (B), A does not wait for B to return any value. If B wants 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 an email 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 it responds 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 assigned to processors by the runtime system, thus creating the runtime system's view of the application as shown in Figure 1.1. The next important decision is determining the execution order of work units on the processors. In the XMAPP model, the availability of messages between work units drives the execution on each processor. In this style of exe-

cution, 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 the 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. 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 multitude 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 thats 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 including:

- Charm++, the widely-used C++-based implementation that this book will focus on
- Charm4Py, which supports the Charm++ model for Python programers
- 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

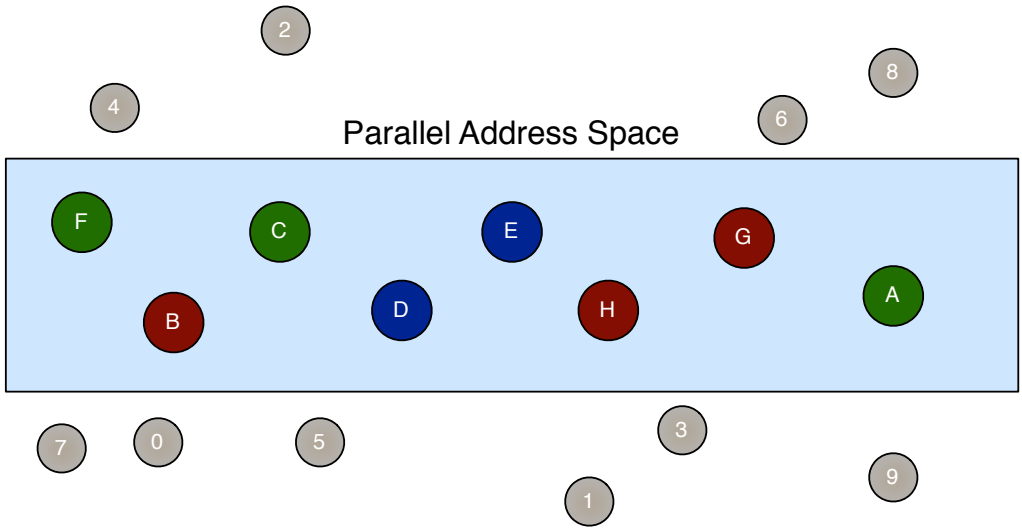

Figure 1.2: Objects in Charm++: certain objects, shown in blue background, are exposed to the runtime system and are controlled by it.

abstractions for expressing certain types of parallel interactions. Here, we briefly introduce the two most commonly used languages: Charm++ and AMPI.

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 connote locality in an additional sense. 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 and enables invocation of methods on remote objects.

A chare is a C++ object which has some additional properties. To make the programmer aware of the potential cost of interacting with remote objects, one 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 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* [1] meaning that 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. An example of dense array is a one dimensional chare array, that has members with indices 0..99. A sparse one-dimensional array example is a chare array consisting of 1000 elements with indices ranging between 1 billion and 2 billion; I.e. it has non-contiguous indices. 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 cannot be an array of basic types such as integers or doubles. Each element of an array is a full-fledged medium-grained chare object that may hold whatever data the application chooses to assign to it: linked lists of particles, arrays of grid points, finite-element subgrids, or trees, for instance. (As a special case, it may *hold* just an integer. But it is still a chare object, with system data, that also includes the integer inside of it. In other words, it inherits from a system-defined Chare class.) 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

[1] For an exception, see a later Chapter 13, but we prefer that a good Charm++ programmer will avoid using those techniques and use them only for exceptional circumstances.

feature of chare arrays is that the elements of an array may be migrated across the processors by the runtime system. Even though the elements may move, the application does not need to be aware of the their locations. When a method is invoked on a certain index of the chare array, the system will find where this element lives, and deliver the method invocation to it, even when the element has recently migrated to a different processor.

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 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 will briefly discuss how Charm++ and its RTS is implemented. This will be useful in understanding the foundations 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].foo(...)` specifies that the method, `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. It shows two processors, each with several chares shown as squares in the figure. Each processor also houses a pool of "messages" where 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].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

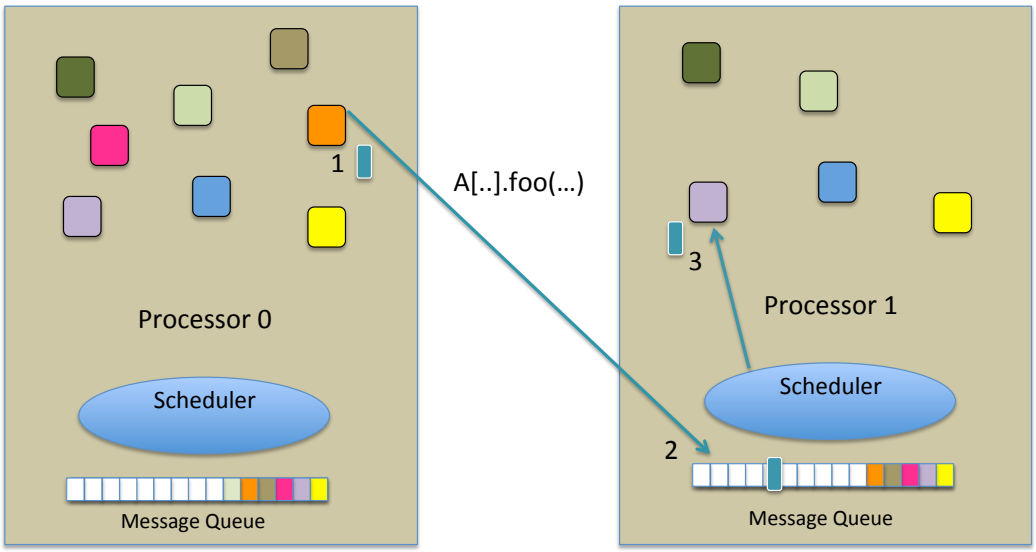

Figure 1.3: Mechanics of method invocation in Charm++: user's call to a proxy is translated to a message that results in invocation of the method on the object eventually.

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 will be delivered on the destination processor (in this case processor 1), and enqueued in the pool of messages on that processor. Later 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 selct 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

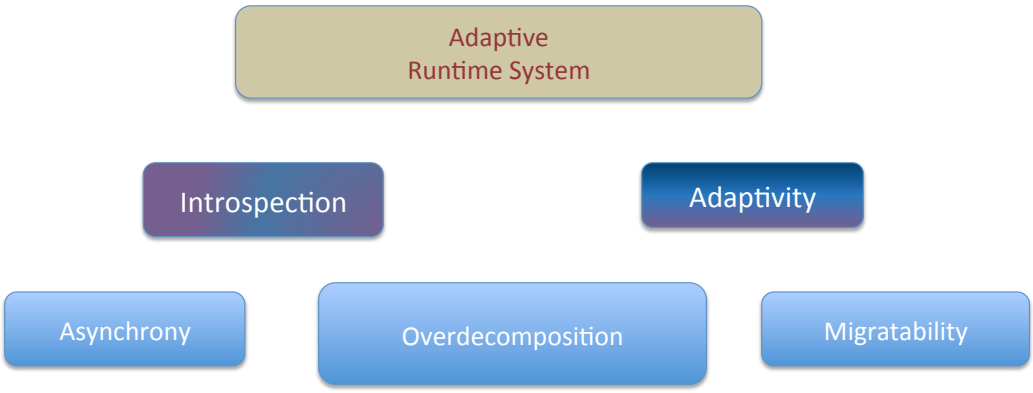

Figure 1.4: Adaptive runtime system of Charm++ and its basis.

imparts the ability to perform "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. As a result, for example, it can prefetch data for those chares from lower-level memory to a higher-level one, such as the scratchpad memory or device memory in some modern accelerators. Compared 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 via the idea of cache hierarchies. This predictability of access in Charm++ is likely to be of increasing use in future architectures.

Communication: The prevalent methods for parallel programming typically lead to a compute-communicate-and-repeat pattern, as shown in Figure 1.5. A programmer 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, which is an increasing challenge given that 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.

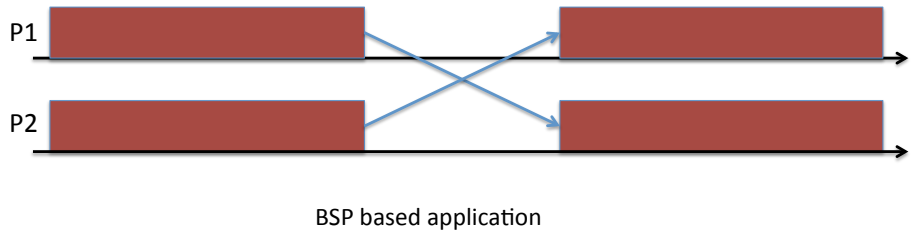

Figure 1.5: An example of compute-communicate pattern in bulk-synchronous parallel programs.

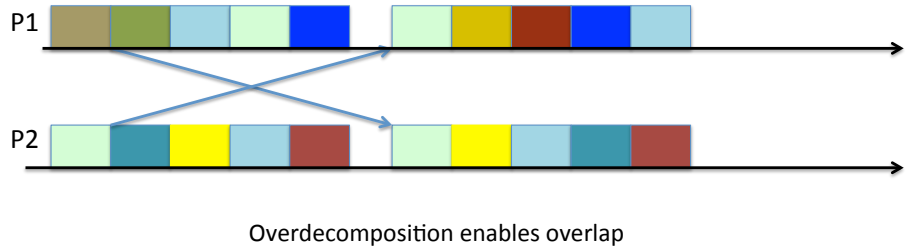

Figure 1.6: Example to demonstrate spread of communication and overlap of computation and communication in Charm++.

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 *enable 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 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 idle time of one module (e.g.

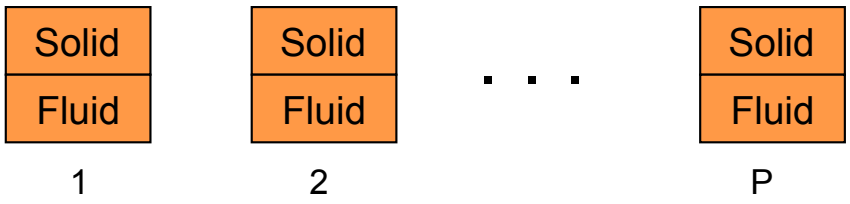

Figure 1.7: Hardware-bound decomposition: irrespective of domain requirements, different entities are divided among P processors, and are arbitrary placed with one another.

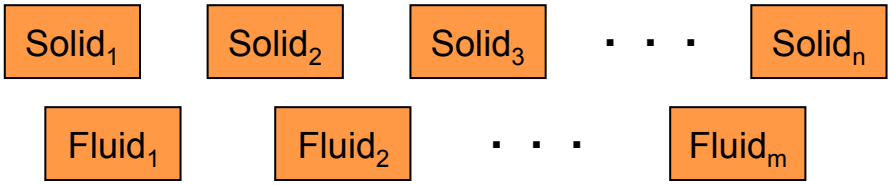

Figure 1.8: Domain driven decomposition in XMAPP: based on the type of entities, they are decomposed differently; the RTS observes these decompositions and place them intelligently.

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 of the processors.

Prioritized scheduling: Scheduling by the RTS is further augmented by adding support for different queuing strategies based on priorities. The programmer assigns priorities to entry method invocations, which are reflected in associated messages. The RTS uses these priorities to reorder scheduling on a processor.

Automatic, dynamic load balancing: The Charm++ runtime system supports measurement-based load balancing. This means that 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 be selected using a simple command-line argument and used in an 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

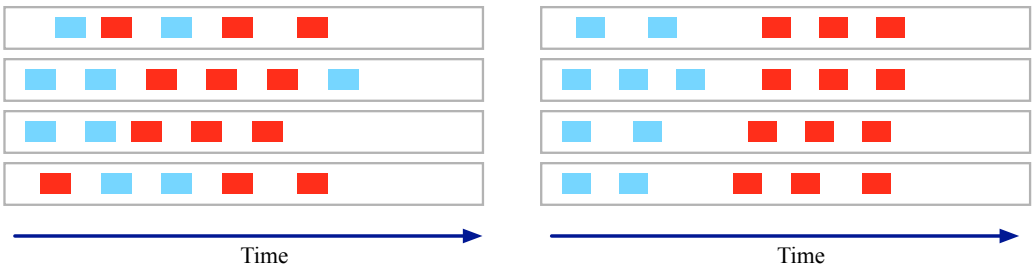

Figure 1.9: RTS controlled message-driven execution enables progress of many independent modules, essentially overlapping overheads of one with useful work of others (left). On the right, an example of time division that will be required in hardware-bound decomposition.

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, and 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.

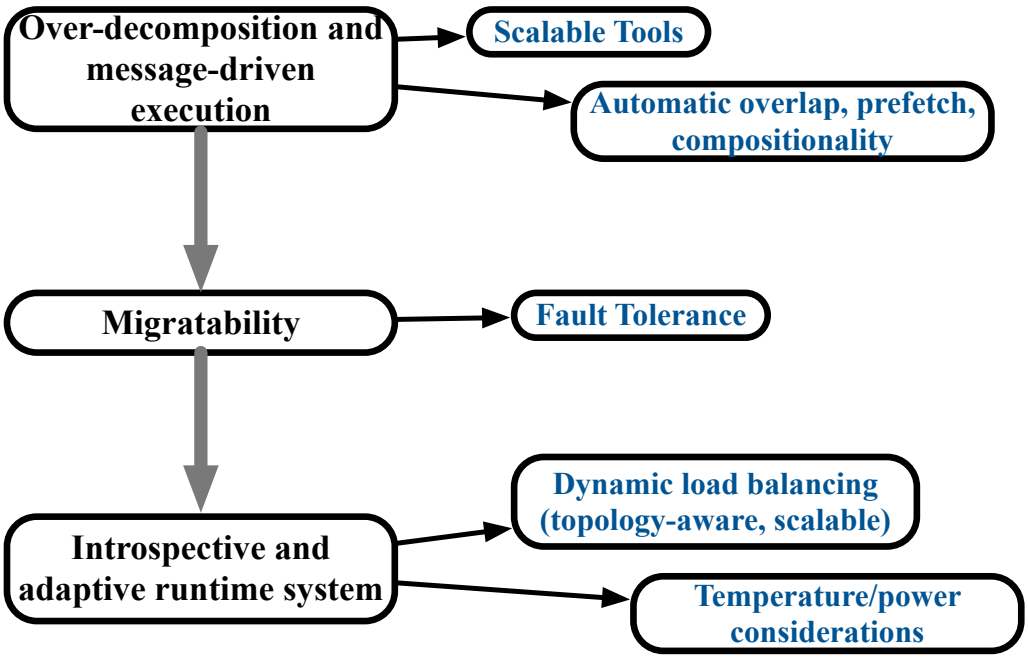

Figure 1.10: Summarizing the benefits of XMAPP languages in general, and Charm++ in particular.

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. You will write programs 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 a special operator for creating new chares (objects) on remote processors.
- A chare class has one or more *entry methods*, which are methods that can be asynchronously invoked from remote processors. The constructor is always an *entry method who is it*.

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. Exactly one module in the program must be designated as a mainmodule. Other modules are designated by the keyword module. All the other modules that you intend to use must be reachable via "extern" module dependencies. . Commonly, a module that contains a main chare is designated as the mainmodule. 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.

```
1  #include <iostream>
2  #include "hello.decl.h"
3
4  /* mainchare */
5  class Origin : public CBase_Origin {
6  public:
7    Origin(CkArgMsg* m) {
8      std::cout << "Hello World!" << std::endl;
9      CkExit();
10   };
11 };
12
13 #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 running this program. Note that you cannot just call exit() for this purpose. 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. (Recall the description of execution and schedulers in section 1.3.1.)

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=netlrts-linux-x86_64. On a 64-bit Mac, you should use version=netlrts-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 seperate 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++ netlrts−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 netlrts-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 to 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++ which handles 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 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.

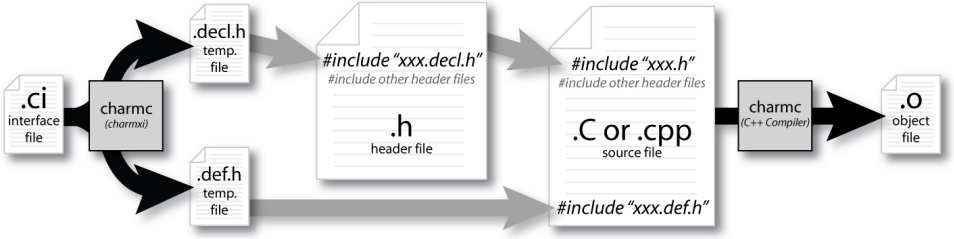

Figure 2.3: Build process of a chare class. If a header file is not used, include the xxx.decl.h file at the top of the .C file in place of the xxx.h file.

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

```
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:

```
./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

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

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 http://charm.cs.illinois.edu/manuals/html/charm++/C.html for further details on running Charm++ programs.

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
```

At this point, you may also benefit from seeing some typical Makefiles in the example programs in the Charm++ distribution. Play Nana lights off

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;
```

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

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

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 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. The 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++ 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 type of parameters [2] and can be overloaded in accordance to C++ standards. In Figure 2.5, constructors of both classes, Interim and Destination, take an integer parameter.

```
1  mainmodule HelloMultiChares {
2    mainchare Origin {
3      entry Origin(CkArgMsg *m);
4    };
5    chare Interim {
6      entry Interim(int x);
7    };
8    chare Destination {
9      entry Destination(int x);
10   };
11 };
```

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

[2]* The parameters are typically deep-copied and do not include references to objects in the calling object. But we will gloss over these details for now. See 4.2)

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 ckNew static method member of the system-generated class as is done in Line 16 of Figure 2.6. The arguments to the ckNew call should be same as the arguments that should be passed to the constructor of the chare class.

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

Lines 28-34 present the definition of the Destination class. Its constructor prints another notification message to the standard output, and calls CkExit to inform the RTS that the user program has completed its tasks. The definition of the auxiliary classes are available in HelloMultiChares.def.h which is included in the C++ file at the end. And, with that, we have a fully functional 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 *even though the constructor of Interim is clearly going to execute after the prints in the Origin constructor*[3].

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

- Use 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.

[3]There are debugging options that make such causally connected prints to appear in order. Refer to the +syncprint command-line option in Charm++ manual.

```
1  #include <stdio.h>
2  #include "HelloMultiChares.decl.h"
3
4  class Origin : public CBase_Origin {
5  public:
6    Origin(CkArgMsg* m) {
7      int key = 1;
8      if(m->argc > 1) {
9        ckout << "<Origin> Hello " << m->argv[1] << "!" << endl;
10       if(m->argc > 2 ) {
11         key = atoi(m->argv[2]);
12       }
13     } else {
14       ckout << "<Origin> Hello World!" << endl;
15     }
16     CProxy_Interim::ckNew(key);
17   }
18 };
19
20 class Interim : public CBase_Interim {
21 public:
22   Interim(int key) {
23     ckout << "<Interim> Got the key from Origin" << endl;
24     CProxy_Destination::ckNew(key);
25   }
26 };
27
28 class Destination : public CBase_Destination {
29 public:
30   Destination(int key) {
31     ckout << "<Destination> Got the key:" << key << endl;
32     CkExit();
33   }
34 };
35
36 #include "HelloMultiChares.def.h"
```

Figure 2.6: Creating multiple chares: the C++ file hello.C.

- Call CkExit() from the last entry method that the program should execute.

2.5 Chare Proxies and Entry Methods

In the examples so far, we have created chares using the ckNew method which led to execution of their constructors. In a more common scenario, the programmer would like to create chares, get a *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 ci file is processed by charmc, proxy classes are declared and defined in the decl.h and 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) ckNew call returns the proxy to the instance that will eventually be created, and 2) every chare class inherits a member named thisProxy from its 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 entry in the ci file. During the processing of the 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 ckNew call, an entry method invocation using a proxy instance tells the Charm++ RTS to execute, at some future time, 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 Main with its constructor being the only entry method. The other class 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 pi will be passed as an argument to it constructor. The class Compute also has another entry method, findArea, that takes two arguments - a double and a 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 findArea method if it should terminate the execution by calling 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 ckNew in a variable sim of type CProxy_Compute. This is an instance of the *proxy* to the created 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

```
1  mainmodule Messaging {
2    mainchare Main {
3      entry Main(CkArgMsg *m);
4    };
5    chare Compute {
6      entry Compute(double y);
7      entry void findArea(double radius, bool);
8    };
9  };
```

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

```
1  #include <stdio.h>
2  #include "Messaging.decl.h"
3
4  /* mainchare */
5  class Main : public CBase_Main {
6  public:
7    Main(CkArgMsg* m) {
8      double pi = 3.1415;
9      CProxy_Compute sim = CProxy_Compute::ckNew(pi);
10     for (int i = 1; i < 10; i++)
11       sim.findArea(i, false);
12     sim.findArea(10, true);
13   };
14 };
15
16 class Compute : public CBase_Compute {
17 private:
18   double y;
19 public:
20   Compute(double pi) {
21     y = pi;
22     ckout << "Hello from a Compute chare running on " << CkMyPe() << endl;
23   }
24
25   void findArea(double r, bool done) {
26     ckout << "Area of a circle of radius " << r << " is " << y*r*r << endl;
27     if (done) CkExit();
28   }
29 };
30
31 #include "Messaging.def.h"
```

Figure 2.8: Method invocation using proxy: the C++ file area.C.

of the real chare and the name of the method in its *envelope*, and send it to the processor that hosts the chare. Using the proxy, in Lines 10-12, several remote invocations are made on the created chare with different input parameters. The last call (Line 12) passes true 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,true)) in the example above may execute before an earlier invocation (e.g. findArea(8,false)), and thus terminate the program prematurely.

There are two reasons 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 7) 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.

```
1  mainmodule Messaging {
2    mainchare Main {
3      entry Main(CkArgMsg *m);
4      entry void doneArea();
5    };
6    chare Compute {
7      entry Compute(double y, CProxy_Main m);
8      entry void findArea(double radius);
9    };
10 };
```

Figure 2.9: Handling out-of-order execution: the interface file areaCorrect.ci.

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 exists 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 findArea queries are executed. To achieve our goal, we modify the example program such that every invocation of findArea makes a remote invocation on the main chare, Main, to inform it about its successful execution.

Figure 2.9 presents the ci file for the correct version of the ongoing example. Three modifications have been made: 1) the main chare, Main, has an entry method called doneArea, which should be called by the instances of Compute class when they execute the findArea method, 2) the constructor of the Compute class has an additional argument of type CProxy_Main which is used to pass a proxy to the main chare, and 3) the second bool argument is no longer required for the findArea entry method.

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

The doneArea method in class Main (Line 17-20) exits once it has been invoked ten times.

```
1  #include <stdio.h>
2  #include "Messaging.decl.h"
3
4  /* mainchare */
5  class Main : public CBase_Main {
6  private:
7    int count;
8  public:
9    Main(CkArgMsg* m) {
10     double pi = 3.1415;
11     CProxy_Compute sim = CProxy_Compute::ckNew(pi, thisProxy);
12     for (int i = 1; i <= 10; i++)
13       sim.findArea(i);
14     count = 10; // wait for 10 responses
15   };
16
17   void doneArea() {
18     count--;
19     if (count == 0) CkExit();
20   }
21 };
22
23 class Compute : public CBase_Compute {
24 private:
25   double y;
26   CProxy_Main mainProxy;
27 public:
28   Compute(double pi, CProxy_Main master) {
29     y = pi;
30     mainProxy = master;
31     ckout << "Hello from a Compute chare running on " << CkMyPe() << endl;
32   }
33
34   void findArea(double r) {
35     ckout << "Area of a circle of radius " << r << " is " << y*r*r << endl;
36     mainProxy.doneArea();
37   }
38 };
39
40 #include "Messaging.def.h"
```

Figure 2.10: Handling out-of-order execution: the C++ file areaCorrect.C.

Notice that even if the ten invocations arrive out of order, the program will exit only after all ten invocations have executed. Also notice that we should not assume that the i^{th} invocation of doneArea will be for radius i, although the for loop in Main::Main might make us think 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, are 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 (as this-¿foo(..) or just foo..)), 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.

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

```
1  mainmodule search {
2    mainchare Main {
3      entry Main(CkArgMsg*);
4    }
5
6    chare Search {
7      entry Search(bool isRoot, unsigned int lo, unsigned int hi,
8                   std::string file, CProxy_Search parent);
9      entry void result(unsigned int line, double min);
10   }
11 }
```

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

the input range is larger than the block size, it will create two new chares responsible for 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 though all $8n$ values and 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.

```
1  #include "search.decl.h"
2
3  #include <string>
4  #include <limits>
5  #include <stdio.h>
6
7  #define BLOCK_SIZE 1000
8
9  struct Main : public CBase_Main {
10   Main(CkArgMsg* msg) {
11     if (msg->argc != 2) {
12       CkPrintf("%s: expecting one argument <filename>\n", msg->argv[0]);
13       CkAbort("");
14     }
15
16     char* filename = msg->argv[1];
17     FILE* file = fopen(filename, "r");
18     if (!file) CkAbort("could not open file\n");
19
20     unsigned lines_count = 0;
21     int ret = fread(&lines_count, sizeof(unsigned), 1, file);
22
23     fclose(file);
24
25     if (ret != 1) {
26       CkAbort("could not read first line properly\n");
27     } else {
28       CkPrintf("Read file, %u lines\n", lines_count);
29     }
30
31     // search range = [0, lines_count)
32     CProxy_Search::ckNew(true, 0, lines_count, std::string(filename), CProxy_Search());
33   }
```

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

```
36 struct Search : public CBase_Search {
37   bool isRoot; CProxy_Search parent; int counter;
38   unsigned int minLine; double minVal;
39
40   Search(bool isRoot_, unsigned int lo, unsigned int hi, std::string filename, CProxy_Search parent_)
41     : parent(parent_), isRoot(isRoot_), counter(0), minLine(0)
42     , minVal(std::numeric_limits<double>::max()) {
43     CkPrintf("Search chare: [%d,%d]\n", hi, lo);
44     if (hi - lo <= BLOCK_SIZE) {
45       FILE* file = fopen(filename.c_str(), "r");
46       if (!file) CkAbort("could not open file\n");
47       if (fseek(file, 4 + lo*8, SEEK_SET) != 0) CkAbort("fseek failed\n");
48       double* val = new double[hi-lo];
49       fread(val, sizeof(double), hi-lo, file);
50       fclose(file);
51       for (int i = 0; i < hi-lo; i++) {
52         if (val[i] < minVal) {
53           minLine = i + lo;
54           minVal = val[i];
55         }
56       }
57       counter++;
58       thisProxy.result(minLine, minVal);
59       delete [] val;
60     } else {
61       const unsigned int mid = (hi - lo)/2 + lo;
62       CProxy_Search::ckNew(false, lo, mid, filename, thisProxy);
63       CProxy_Search::ckNew(false, mid, hi, filename, thisProxy);
64     }
65   }
66
67   void result(unsigned int line, double min) {
68     counter++;
69     if (min < minVal) {
70       minLine = line;
71       minVal = min;
72     }
73     if (counter == 2) {
74       if (isRoot) {
75         CkPrintf("Found minimum value %f on line %d\n", minVal, minLine);
76         CkExit();
77       } else {
78         parent.result(minLine, minVal);
79       }
80     }
81   }
82 };
83
84 #include "search.def.h"
```

Figure 2.13: Recursive chare file search: the C++ file (Part 2: Search chare) search.cpp.

As we decrease the BLOCK_SIZE we enable the runtime system to exercise more flexibility in 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 overage 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 five microseconds). 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:

$T_p = \max \{g, T_1/P\}$

$T_p = \max \{ \text{ g}, \frac{T(1+\frac{t_o}{g})}{P} \}$

The max operator arises here because one cannot execute the program any faster than the 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

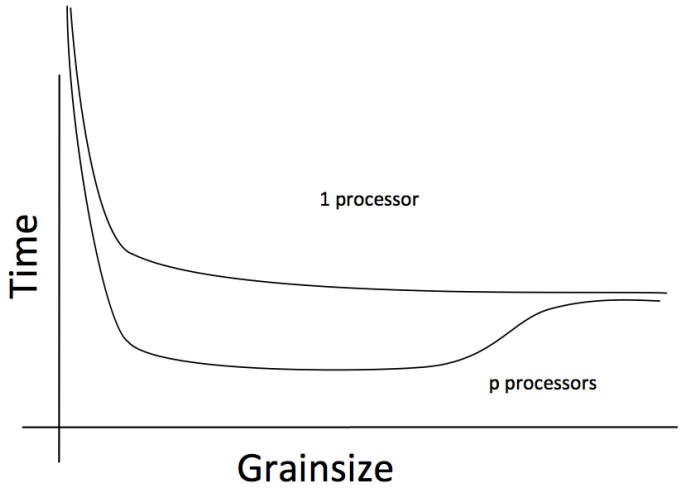

Figure 2.14: Impact of grainsize on total execution time.

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 μ, thats about 60 μs!

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..

Although the X-axis in this plot is the number of chares per core, it is better to think in terms of computation per message. Typically, entry methods that take tens of microseconds on the average do not add significan overhead.

().

So, our conclusion is: the overhead of overdecomposition is not significant for many real-

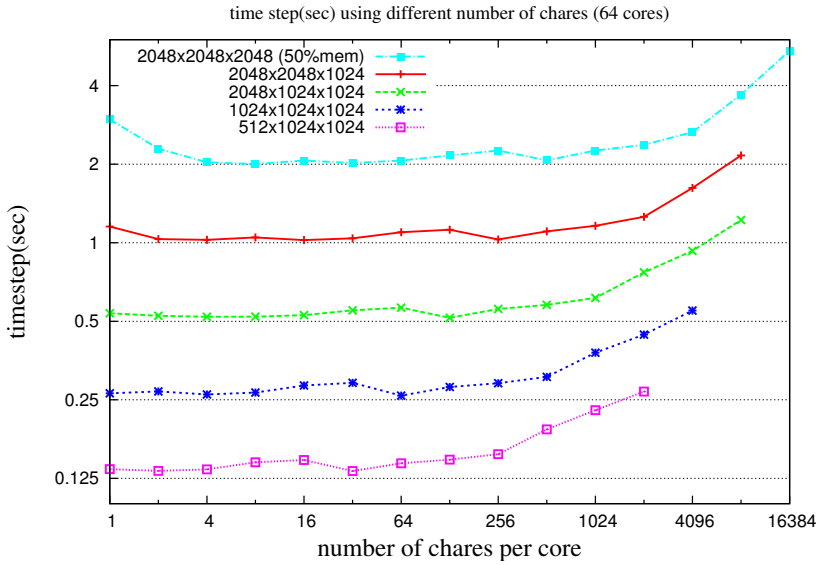

Figure 2.15: Effect of grainsize on a simple parallel 3D-stencil.

istic 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 Other ways of fixing the findArea program

The program **??** was wrong because the entry method invocation with "done" as the second parameter might arrive at the destination chare before some of the earlier invocations. We fixed it somewhat expensively, by requiring every invocation to be acked via a return invocation going to the main chare. This doubles the amount of communication. In this exercise, you will develop and code other alternate solutions to this problem. These alternate solutions represent patterns that you will encounter in more complex settings, so its worth paying attention to each.

Counting up

Let us assume that the main chare knows, before constructing the Compute chare, how many findArea invocations will be sent to it. Add a constructor argument to pass this information to Compute, have Compute store the expected number of messages, and modify the test in findArea() to count up to it before calling CkExit().

Sending Count with the last call

In a general situation, the main chare may not know how many times it is going to invoke findArea() when it creates the Compute chare. But, let us assume, it does know when it is done calling, as in the original code, where we signified that by passing "done" as a boolean. You can deal with that by passing the real count in the last call. Of course, all the preceding calls must still include this count parameter, but it can have a special dummy value, such as -1, which will signify that this was not the last invocation. Modify the code to make this idea work.

2.9.2 Calculating Pi by Throwing Darts

Many mathematical computations use the value of π, but finding the value of π is an interesting task by itself. This exercise is to write a simple parallel program to compute the value of π based on one of its know usage. The area of a circle of radius 1 unit is π, whereas the area of a square of size 2 units, within which such a circle is inscribed, is 4 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.

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

Assume that numTrials has to be a large value in order to get a reasonably correct value of π. 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 π. 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 π 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.

2.9.3 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:

```
fib(n) = if (n < 2) n else fib(n−1) + 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 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 THRESHOLD, calculate fib(n) using a sequential version of the algorithm.

2.9.4 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 `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) don't let the loop run all the way to "number-1").

```
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 is a way to imrpove 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, generatee them by concatenating 2 32 bit random numbers.

2.9.5 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.

Chapter 3

Chare Arrays: Indexed Collections of Chares

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 use 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

```
1  mainmodule arrayRing {
2
3    mainchare Main {
4      entry Main();
5      entry void ringFinished();
6    };
7
8    array [1D] Ring {
9      entry Ring(CProxy_Main mp, int rs);
10     entry void doSomething(int elementsLeft, int tripsLeft, int fromIndex, int fromPE);
11   };
12
13 }
```

Figure 3.1: Single Ring: interface file arrayRing.ci

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 the 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.

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.

```
6  class Main : public CBase_Main {
7   public:
8    Main(CkMigrateMessage *msg) { }
9    Main(CkArgMsg* msg) {
10     int ringSize = atoi(msg->argv[1]);
11     int tripCount = atoi(msg->argv[2]);
12     delete msg; // Done with message
13
14     // Display some information to the user about this run
15     CkPrintf("\"Array Ring (Single)\" Program\n");
16     CkPrintf(" ringSize = %d, tripCount = %d, #PEs() = %d\n", ringSize, tripCount, CkNumPes());
17
18     // Create the chare array and start at a random element
19     CProxy_Ring ring = CProxy_Ring::ckNew(thisProxy, ringSize, ringSize);
20     srand(time(NULL)); // Initialize random number generator
21     ring(rand() % ringSize).doSomething(ringSize, tripCount, -1, -1);
22   }
23
24   void ringFinished() { CkExit(); }
25 };
```

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.

```
28 class Ring : public CBase_Ring {
29  private:
30   CProxy_Main mainProxy; // Proxy object for the main chare
31   int ringSize; // Number of elements in the ring
32  public:
33   Ring(CkMigrateMessage *msg) { }
34   Ring(CProxy_Main mp, int rs) {
35     mainProxy = mp;
36     ringSize = rs;
37   }
38
39   inline int nextI() { return ((thisIndex + 1) % ringSize); }
40
41   void doSomething(int elementsLeft, int tripsLeft, int fromIndex, int fromPE) {
42     // Do something (display some text for the user)
43     printf("Ring[%d](%d): tripsLeft = %d, from [%d](%d)\n", thisIndex, CkMyPe(), tripsLeft,
           fromIndex, fromPE);
44
45     // Send message to continue traversals or notify main
46     if (elementsLeft > 1) { // elements left in traversal
47       thisProxy(nextI()).doSomething(elementsLeft - 1, tripsLeft, thisIndex, CkMyPe());
48     } else if (tripsLeft > 1) { // start next traversal
49       thisProxy(nextI()).doSomething(ringSize, tripsLeft - 1, thisIndex, CkMyPe());
50     } else { // otherwise, all traversals finished
51       mainProxy.ringFinished();
52     }
53   }
54 };
```

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

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 reseting 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 it's 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).

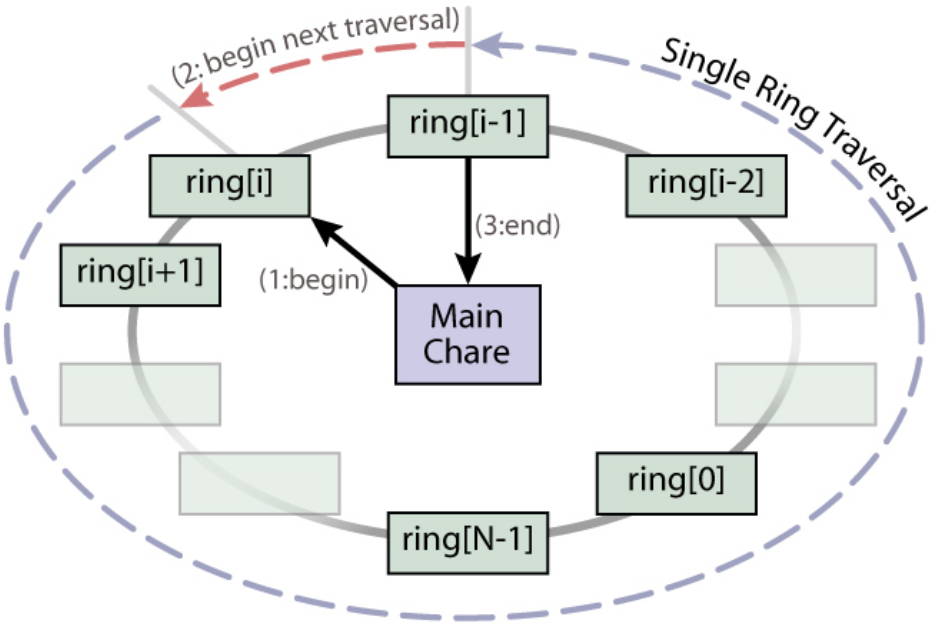

Figure 3.4: Control flow of Single Ring example program. There are N array elements in the ring, and index i is the element chosen by the mainchare to begin the ring traversals.

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.

```
$ ./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

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. For illustrative purposes, a skip of 1 is shown. 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

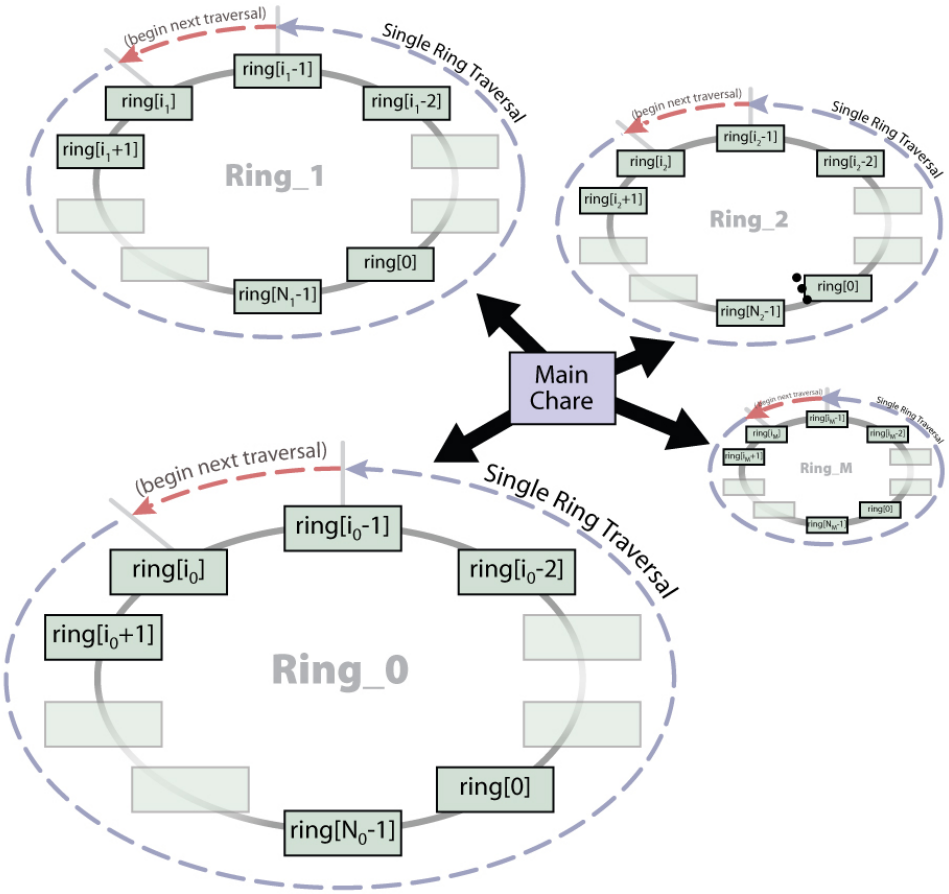

Figure 3.6: Control flow of Multiple Ring

```
1  mainmodule multiRing_Main {
2
3    // Declare chare objects/collections for the Main module
4    mainchare Main {
5      entry Main();
6      entry void ringFinished();
7    };
8
9    // Include the Ring object's module
10   extern module multiRing_Ring;
11
12 }
```

Figure 3.7: Multi Rings program: mainchare class interface file

```
1  module multiRing_Ring {
2
3    // Include the Main object's module (main proxy reference)
4    extern module multiRing_Main;
5
6    // Declare chare objects/collections for the Ring module
7    array [1D] Ring {
8      entry Ring(CProxy_Main mp, int rs, int rID);
9      entry void doSomething(int elementsLeft, int tripsLeft, int fromIndex, int fromPE);
10   };
11
12 }
```

Figure 3.8: Multi Rings program: ring chare array class interface file

```
8  class Main : public CBase_Main {
9    int numRings = 0;
10  public:
11   Main(CkMigrateMessage *msg) { }
12   Main(CkArgMsg* msg) {
13     int *ringSize = NULL, *tripCount = NULL;
14     processCommandLine(msg, &numRings, &ringSize, &tripCount);
15
16     CkPrintf("\"Array Ring (Multi)\" Program\n");
17     CkPrintf(" numRings = %d, #Pes() = %d\n", numRings, CkNumPes());
18     for (int i = 0; i < numRings; i++)
19       CkPrintf(" Ring_%d : ringSize = %d, tripCount = %d\n", i, ringSize[i], tripCount[i]);
20
21     // Create the rings
22     for (int i = 0; i < numRings; i++) {
23       CProxy_Ring ring = CProxy_Ring::ckNew(thisProxy, ringSize[i], i, ringSize[i]);
24       ring(rand() % ringSize[i]).doSomething(ringSize[i], tripCount[i], -1, -1);
25     }
26   }
27
28   void processCommandLine(CkArgMsg* msg, int* nr, int** rs, int** tc);
29   void printUsage(const char* const errStr, char* appName);
30
31   void ringFinished() {
32     static int finishedCount = 0;
33     if ((++finishedCount) >= numRings) { CkExit(); }
34   }
35 };
```

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

of work among different modules.

```
5  class Ring : public CBase_Ring {
6   private:
7    CProxy_Main mainProxy; // Proxy object for the main chare
8    int ringSize; // Number of elements in the ring
9    int ringID; // ID value for this ring
10
11  public:
12   Ring(CkMigrateMessage *msg) { }
13   Ring(CProxy_Main mp, int rs, int rID) : mainProxy(mp), ringSize(rs), ringID(rID) { }
14
15   int nextI(int s) { return ((thisIndex+s) % ringSize); }
16
17   void doSomething(int elementsLeft, int tripsLeft, int fromIndex, int fromPE) {
18
19     // Do something (display some text for the user)
20     printf("Ring_%d[%d](%d): tripsLeft = %d, from [%d](%d)\n", ringID, thisIndex, CkMyPe(),
           tripsLeft, fromIndex, fromPE);
21
22     // Send message to continue traversals or notify main
23     if (elementsLeft > 1) { // elements left in traversal
24       int skipAmount = (rand() % elementsLeft) + 1;
25       thisProxy(nextI(skipAmount)).doSomething( elementsLeft - skipAmount, tripsLeft, thisIndex,
             CkMyPe());
26     } else if (tripsLeft > 1) {
27       thisProxy(nextI(1)).doSomething( ringSize, tripsLeft - 1, thisIndex, CkMyPe());
28     } else {
29       mainProxy.ringFinished();
30     }
31   }
32 };
```

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.

```
$ ./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.

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

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

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

Figure 3.12: 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 $\otimes$ is said to be commutative if $A \otimes B = B \otimes A$. Integer multiplication is an example of a commutative operation ($2*3 = 3*2$). 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 $\oplus$ is said to be associative if $(A \otimes B) \otimes C = A \otimes (B \otimes C)$. Integer addition is an example of an associative operation ($(5+3)+2 = 5+(3+2)$). 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, $\otimes$, is a value such that $a \otimes I = a$. For example, zero is the identity value of addition ($a+0=a$), and one is the identity value of multiplication ($a*1=a$).

```
1  // Perform a reduction on an interger array
2  // r = a[0] + a[1] + ... + a[len−1]
3  int reduceArray(int *a, int len) {
4    int r = 0;
5    for (int i = 0; i < len; i++)
6      r += a[i];
7    return r;
8  }
9
10 // Perform a reduction on an integer array by splitting into two halves
11 // r = (a[0] + ... + a[len / 2]) + (a[len / 2 + 1] + ... a[len − 1])
12 int reduceArraySplit(int *a, int len) {
13   // add the values in the first half of 'a'
14   int r1 = 0;
15   for (int i = 0; i < len / 2; i++) {
16     r1 += a[i];
17
18   // add the values in the second half of 'a'
19   int r2 = 0;
20   for (int i = len / 2 + 1; i < len; i++)
21     r2 += a[i];
22
23   return r1 + r2;
24 }
```

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:

```
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:

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

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

```
CkCallback cb(CkReductionTarget(ClassName, entryMethod), proxyToClassName);
```

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 in to 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.

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

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

(a) reduction.ci

```
#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"
```

(b) reduction.cc

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 naïve 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:

for i = 1 to N 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 in $PlgP$, 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 lgP 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 recieves) 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, lets 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"

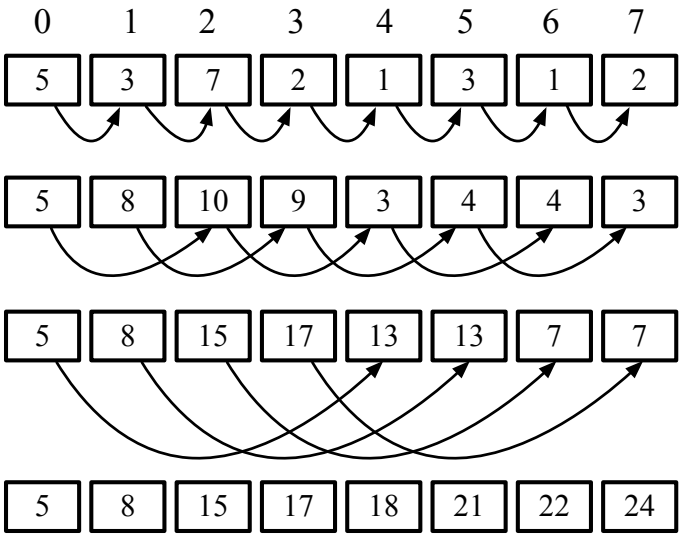

Figure 3.15: At each stage, the distance that each chare's value is sent doubles.

```
1  mainmodule prefix {
2    readonly CProxy_Main mainProxy;
3    readonly int numElements;
4    readonly CProxy_Prefix prefixArray;
5
6    mainchare Main {
7      entry Main(CkArgMsg*);
8      entry [reductiontarget] void done();
9    };
10
11   array [1D] Prefix {
12     entry Prefix();
13     entry void step(int);
14     entry void passValue(int stage, int value);
15   };
16 };
```

Figure 3.16: prefixSum.ci.

entry 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 passed 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. [5]. 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.

```
1  #include "prefix.decl.h"
2  #include <math.h>
3
4  /*readonly*/ CProxy_Main mainProxy;
5  /*readonly*/ CProxy_Prefix prefixArray;
6  /*readonly*/ int numElements;
7
8  class Main : public CBase_Main {
9  public:
10   Main(CkArgMsg* msg) {
11     mainProxy = thisProxy;
12     numElements = 8; //default
13     if(msg->argc > 1) numElements = atoi(msg->argv[1]);
14     delete msg;
15     prefixArray = CProxy_Prefix::ckNew(numElements);
16   }
17   Main(CkMigrateMessage* msg) { };
18   void done() { CkExit(); }
19 };
20
21 class Prefix : public CBase_Prefix {
22 public:
23   int *valueBuf, *flagBuf, value, stage, numStages;
24   Prefix() : stage(0) {
25     numStages = log2(numElements);
26     valueBuf = (int*)malloc(numStages*sizeof(int));
27     flagBuf = (int*)malloc(numStages*sizeof(int));
28     srand(thisIndex);
29     value = rand()%10 + 1;
30     //CkPrintf("Chare %d generated value %d\n", thisIndex, value);
31     step(value);
32   }
33   Prefix(CkMigrateMessage*) { };
34
35   void step(int value) {
36     if(stage >= numStages) {
37       //CkPrintf("Chare %d on PE %d finished with final value %d\n", thisIndex, CkMyPe(), value);
38       CkCallback cb(CkReductionTarget(Main, done), mainProxy);
39       contribute(sizeof(int), &value, CkReduction::sum_int, cb);
40     }
41     else {
42       int sendIndex = thisIndex + (1 << stage);
43       if(sendIndex < numElements) thisProxy[sendIndex].passValue(stage, value);
44       if(flagBuf[stage] == 1) { updateValue(); }
45       else if(thisIndex - (1 << stage) < 0) {
46         stage++;
47         step(value);
48       }
49     }
50   }
51
52   void passValue(int incoming_stage, int incoming_value) {
53     flagBuf[incoming_stage] = 1;
54     valueBuf[incoming_stage] = incoming_value;
55     if(flagBuf[stage] == 1) { updateValue(); }
56   }
```

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 than required 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 *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

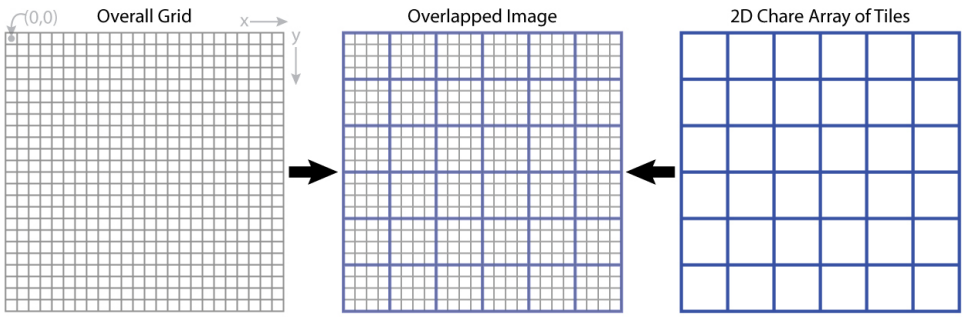

Figure 3.18: Decomposition of 2-D stencil grid into tiles.

tile that will be used as input to the element calculations for the neighboring tile's border element's calculations. *Ghost* is used because once the data arrives, it can only be read, 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 that 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 straight forward application of a reduction. Remember that a reduction can preform any commutative and associative mathematical operation. In this case, the mathematical operation of *maximum* has both of these mathematical properties and thus can be used in a reduction to calculate the global maximum value change seen across all the tiles for any given timestep.

Once each tile 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 recieve four ghost messages, even if they are on the edge of the 2-D chare array.

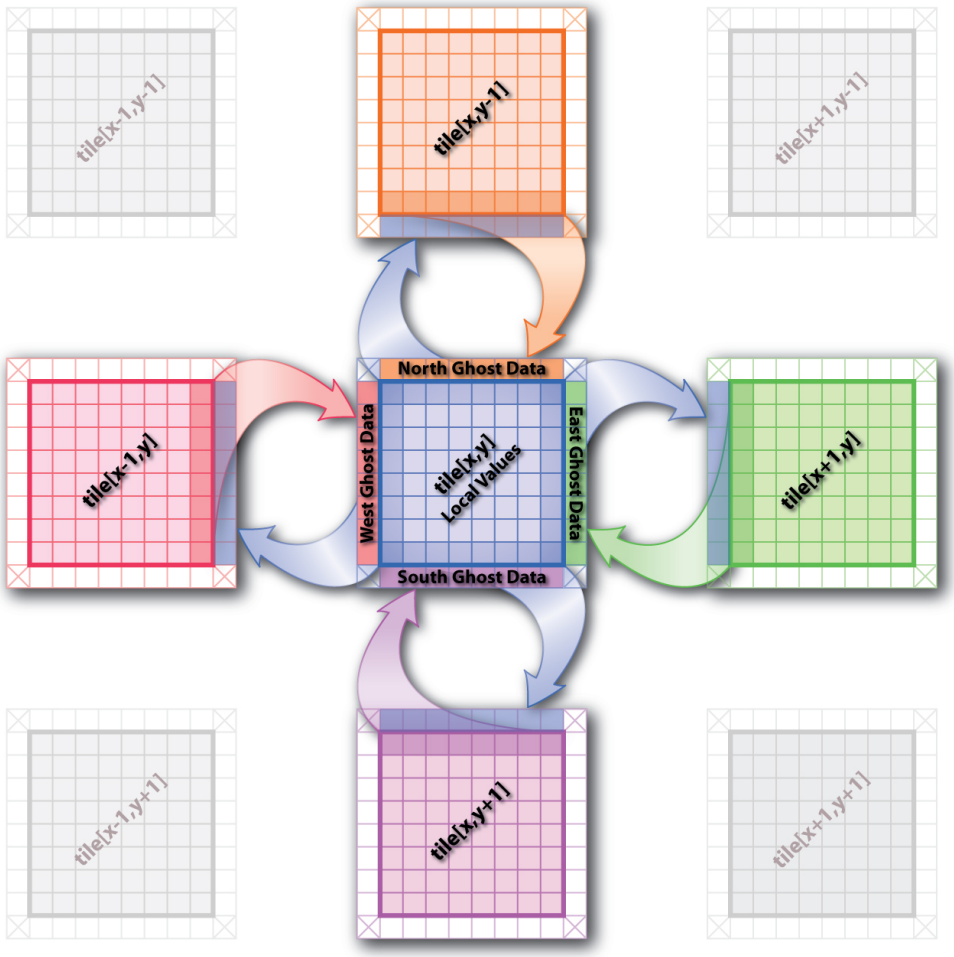

Figure 3.19: Communication pattern of tiles in the 2-D stencil application.

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

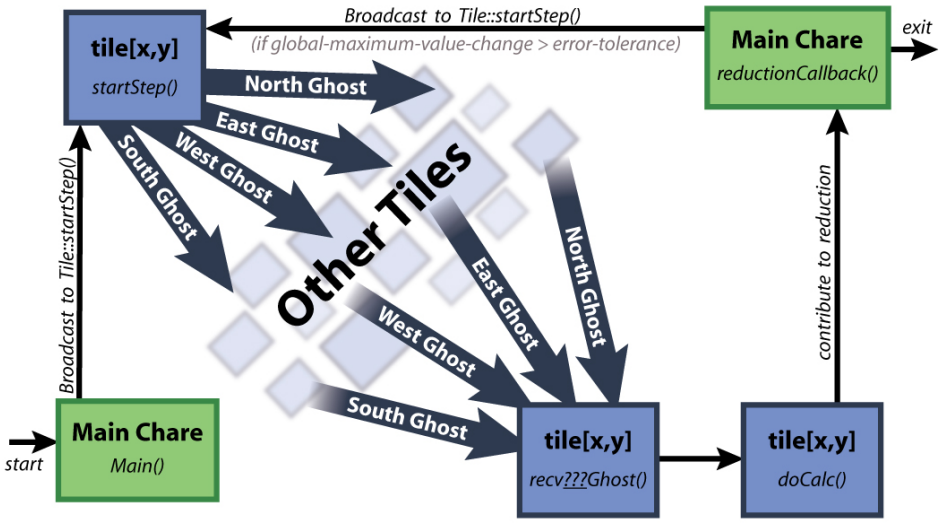

Figure 3.20: Logical flow of 2D 5-point stencil execution from the point of view of a single tile object (*tile[x,y]*).

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.

```
1  mainmodule main {
2    readonly float targetDiff;
3    readonly int gridWidth;
4    readonly int gridHeight;
5    readonly int tileWidth;
6    readonly int tileHeight;
7
8    mainchare Main {
9      entry Main(CkArgMsg*);
10     entry [reductiontarget] void reductionCallback(float maxStepDiff);
11   };
12
13   extern module tile;
14 }
```

```
14 class Main : public CBase_Main {
15  private:
16   CProxy_Tile grid;
17   void processCommandLine(const CkArgMsg * const msg);
18   void displayUsage(const CkArgMsg * const msg);
19   void displayHeader();
20
21  public:
22   Main(CkMigrateMessage* msg);
23   Main(CkArgMsg* msg);
24   void reductionCallback(float maxStepDiff);
25 };
```

Figure 3.21: The interface and header files for the main chare object in the 2-D stencil application.

```
18 Main::Main(CkArgMsg* msg) {
19   processCommandLine(msg);
20   displayHeader();
21   delete msg;
22
23   // Create the grid tiles and set the reduction client
24   grid = CProxy_Tile::ckNew(gridWidth, gridHeight);
25   CkCallback cb(CkReductionTarget(Main, reductionCallback), thisProxy);
26   grid.ckSetReductionClient(&cb);
27
28   // Start the first step
29   grid.startStep();
30 }
```

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

```
38 void Tile::startStep() {
39   const int thisX = thisIndex.x;
40   const int thisY = thisIndex.y;
41
42   // Send to the north (target tile receives from the south)
43   float* northGhost = tileData + NORTH_OFFSET + TILE_Y_STEP;
44   thisProxy(thisX, thisY > 0 ? thisY - 1 : gridHeight - 1).recvSouthGhost(northGhost, tileWidth);
45
46   // Send to the south (target tile recieves from the north)
47   float* southData = tileData + SOUTH_OFFSET - TILE_Y_STEP;
48   thisProxy(thisX, thisY < gridHeight - 1 ? thisY + 1 : 0).recvNorthGhost(southData, tileWidth);
49
50   // Send to the west (target tile recieves from the east)
51   for (int i = 0; i < tileHeight; i++)
52     scratchData[i] = tileData[WEST_OFFSET + TILE_X_STEP + (TILE_Y_STEP * i)];
53   thisProxy(thisX > 0 ? thisX - 1 : gridWidth - 1, thisY).recvEastGhost(scratchData, tileHeight);
54
55   // Send to the east (target tile recieves from the west)
56   for (int i = 0; i < tileHeight; i++)
57     scratchData[i] = tileData[EAST_OFFSET - TILE_X_STEP + (TILE_Y_STEP * i)];
58   thisProxy(thisX < gridWidth - 1 ? thisX + 1 : 0, thisY).recvWestGhost(scratchData, tileHeight);
59
60   countEvent();
61 }
```

Figure 3.23: The Tile::startStep() method in tile.C.

```
64 void Tile::recvNorthGhost(float* ghostData, int dataLen) {
65   memcpy(tileData + NORTH_OFFSET, ghostData, dataLen * sizeof(float));
66   countEvent();
67 }
68
69 void Tile::recvSouthGhost(float* ghostData, int dataLen) {
70   memcpy(tileData + SOUTH_OFFSET, ghostData, dataLen * sizeof(float));
71   countEvent();
72 }
73
74 void Tile::recvWestGhost(float* ghostData, int dataLen) {
75   for (int i = 0; i < tileHeight; i++)
76     tileData[WEST_OFFSET + (TILE_Y_STEP * i)] = ghostData[i];
77   countEvent();
78 }
79
80 void Tile::recvEastGhost(float* ghostData, int dataLen) {
81   for (int i = 0; i < tileHeight; i++)
82     tileData[EAST_OFFSET + (TILE_Y_STEP * i)] = ghostData[i];
83   countEvent();
84 }
```

Figure 3.24: Entry methods that receive ghost data in the Tile chare class in tile.C.

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 recieve 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

```
87 void Tile::countEvent() {
88   if ((++eventCounter) >= 5) {
89     eventCounter = 0;
90     doCalc();
91   }
92 }
```

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 straight forward. 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.[7] 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

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

```
94  void Tile::doCalc() {
95    float maxDiff = ((float)(0.0));
96
97    // Perform the 5−point calc on all elements in the tile
98    for (int y = 1; y < tileHeight + 1; y++)
99      for (int x = (y == 1 ? 2 : 1); x < tileWidth + 1; x++) {
100       float origVal = tileData[XY_TO_I(x,y)];
101       float newVal = (origVal + tileData[XY_TO_I(x+1, y)] +
102         tileData[XY_TO_I(x−1, y)] + tileData[XY_TO_I(x, y+1)]
103         + tileData[XY_TO_I(x, y−1)]) * ((float)(0.2));
104       scratchData[XY_TO_I(x,y)] = newVal;
105       float diff = fabsf(origVal − newVal);
106       maxDiff = fmax(maxDiff, diff);
107     }
108
109   // Contribute to the step's reduction
110   contribute(sizeof(float), &maxDiff, CkReduction::max_float);
111
112   // Swap the data buffer pointers and enforce constants
113   float* tmp = tileData;
114   tileData = scratchData;
115   scratchData = tmp;
116   enforceConstants();
117 }
```

```
138 void Tile::enforceConstants() {
139   if (tileData != NULL) { tileData[XY_TO_I(1,1)] = MAX_VAL; }
140 }
```

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

```
32 // Function each grid element will call as a step finishes
33 void Main::reductionCallback(float maxStepDiff) {
34   // Display this step's maximum difference to the user
35   static int numStepsCompleted = 0;
36   CkPrintf("Step %d: %f\n", ++numStepsCompleted, maxStepDiff);
37
38   // Check to see if the difference is small enough that the
39   // application can complete
40   if (maxStepDiff <= targetDiff) {
41     CkExit(); // Program finished
42   } else {
43     grid.startStep(); // Another step needs to be run
44   }
45 }
```

Figure 3.27: The main chare object's reduction callback function. This function is called when the reduction across the entire grid completes for each timestep.

element values for the tile. The tileData and scratchData pointers are then swapped. The call to Tile::enforceConstants enforces the constant values since these elements are skipped by the for loops in Tile::doCalc and no assumptions are made about the initial contents of the scratchData when Tile::doCalc begins.

Figure 3.27 contains the reduction callback function use by the tile array. The code compares the global maximum value change, computed from the reduction, with the error tolerance targetDiff. If the maximum value change is larger than the error tolerance, the main chare initiates another timestep by once again invoking the Tile::startStep entry method on all the chare array elements (i.e. broadcasting to the chare array). Otherwise, it calls CkExit(), thereby exiting the application

Figure 3.28 contains the output from an example run of the 2-D 5-point stencil program using four processors.

3.6 Pitfalls

Once you internalize the basic concepts of Charm++, itbecomes, for most people, a highly intuitive programming model. But it takes some effort to overcome misunderstandings based on ingrained "biases" that shape one's (mis)understanding of the language. It may be useful to look at some example programs to clarify those. But first, let us restate the "basic concepts" that you need to internalize. The list below is not independent, in that some concepts can be erived from others, but it is helpful to restate them in different ways.
caveat: Many advanced programming methods that will be discussed in later chapters allow

```
$./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.

the programmer to circumvent some of the issues discussed below for the sake of ease of expression. But it is important to internalize the basic distributed memory message-driven execution model we learned. sofar, in order to understand how and when. to use those advanced constructs correctly and effectively.

1. *You cannot access data members from another chare.* For one thing, you only have a "proxy" to another chare, not a pointer. (and if you were to acquire a pointer, because afterall a chare is a C++ object, it will not be valid across processes/nodes, and even when it is within your own process, the baseline semantics requires you not access such variables for avoiding race conditions). The only way to get information that another chare holds is to get it to send it to you by invoking an entry method on your chare.

2. The only kind of global variable allowed is a readonly variable, and one that is declared as such in the .ci file. Again, you can see why this is so if you understand the execution model. Since there are multiple processes on multiple nodes that house the chares, modifications to a variable modified by one chare will only be visible to other chares on the same process, and since you don't know where another chare. is located, you should not rely on communicating via changing global variables.

3. Don't elliminate asynchrony by sequentializing actions rigidly. That can lead to loss of performance or even deadlocks. Asynchronoy is your friend as well as your enemy.

4. Actions (entry method invocations) that are pending can happen in any order under the control of the runtime system and its scheduler. Do not fall into the trap of programming for (i.e. assuming) some particular expected order.

5. No data members of a chare can change their value "under your feet" as you are executing one of the entry methods (unless that method itself is changing it). In particular, if you ask another chare to send a value to one. ofyour entry methods, you cannot expect the value. tohave arrived a few statements later within the same method.

3.7 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 `reductiontarget` method in main chare, which just prints all the added up values from each iteration. Take care to ensure each element contributes exactly once in each reduction. Observe, via projections and by inspecting your code, how reduction progress is happening interspersed with ring messages.

2. Give a collection M 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 typically 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 calclulate, 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 rsponding 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 fullfied 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 its 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 invication's cost $\alpha + n.\beta$, where α is a fixed cost, and β is a per-byte cost. For the sake of intuition, you may assume α is about a thousand times larger than β, 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 mnumber of physical processors.

   B. Code one of the algorithms you identified above.

6. **Communication Granularity with client-server**

   Create a chare array A with 2 chares. A[0] is the producer and A[1] is the consumer. Add entry methods `makeRequests`, `request` and `response` to A. The constructor of A, if it is on rank A[0], asynchronously invokes its own makerequest method. Use a readonly int variable batchSize or just K for brevity. N, the total number of numbers, is another paramer. Both of these are to be read in from command line.

   makeRequest repeatedly create batches of K random double precision numbers, N/K times, and invokes A[1].request(..) with that data. A[1]'s reqeust method calculates square of each number provided and sends the array of size K of these squares to

response method of A[0], which simply discards them (if you want, you can add code to make sure A[1] correctly created the squares.

You may assume K evenly divides N for the initial exercise (but see part C).

Once the code is running correctly, experiment with different values of K for the same N. Use $N = 2^30 for this part. Plot performance (i.e. execution time measured from the main chare)$

(a) when the chares are on the same PE

(b) when the two chares are different PEs on the same process

(c) when the two chares are on different processes within the same physical node (host)

(d) when the two chares are on different PEs on different physical nodes

Learn what you need to about proper way of time measurement, initial mapping, and SMP build.

This exercise is to help your intuition about performance issues and communication size.

7. **Efficient Median Finding:**

Let us assume you have P chares in

create a chare array A with P chares. In its constructor, each chare A[i] generates M random double precision numbers with poisson distribution with mean = (100.0*i)/N, and stores them in a local vector X.

Write an efficient program to find the median of all the numbers stored in the X vectors on all chares. No communication should be more than 1024 bytes, irrespecive of the value of P and M. You may use reductions and broadcasts in addition to point to point messages.

(An asynchronous invocation of an entry method is also called a message, and its size is roughly the size of all the parameters to the entry method.. We say "roughly" because there is also an envelope containing metadata that is sent along with each entry method invocation.

Chapter 4

PUP and PUP::er, or how to serialize data

This chapter and the next two chapters are connected by the theme of migratability, and facilitating data movement.

PUP stands for pack-and-unpack, or in other words, data serialization and deserialization. Data that we need to communicate or transfer is typically organized as multiple individually contiguous but collectively non-contiguous blocks.

1. Parameter-marshalled entry methods
2. Object serialization using Charm++'s PUP framework.
3. How to use dynamic load balancing in Charm++ programs
4. How to use checkpoint/restart and automatic fault tolerance

4.1 Chapter: PUP and data serialization

For a Charm++ program running on multiple hosts, the Charm++ runtime will typically need to send data across hosts. This may be to convey the parameters of an entry method invocation or to migrate chares for load balancing. You will need to help the runtime "package" your data for such transfers. Keep in mind that the the charmc compiler only processes your ".ci" files, and it has no idea about the class definitions in your .h or .C or .cpp files. So, it has to generate code for processing your data structures based solely on information you provide in the interface file/s.

Consider a situation where you wish to send a data structure of type Vec3d as a parameter to an entry method E. Vec3d is defined as below, in your .h file.

```
struct vec3d {double x, y, z;};
```

the entry method declaration for E, in your .ci file, looks like this:

```
entry void E(int m, vec3d v, float A[m]);
```

Well, the Charm compiler knows how to package m, and the array A, but it has no idea what to do with the middle parameter, because it does not know what vec3d is.

In this and other simple cases like this, you can simply tell the Charm system to pack or unpack this structure as a bunch of bytes. You do that by using a system-provided macro in your .h file.

```
PUPBytes(vec3d)
```

This will work, as long as you are running on a homogeneous machine where the representation of a "double" does not change from host to host. (POSTPONE THIS TEXT?? On most clusters today, this is a reasonable assumption. But if you want a more general solution, you should use the method described later, by defining the "|" operator).

Of course, you may use arrays of such structures, or structures containing arrays of such structures, in a similar manner. As long as the system can find the size of the structures involved, by simply calling "sizeof" operator, this mechanism (i.e. using the PUPBytes macro) will work correctly.

More complex situations require a general solution. To understand this, consider what the runtime system (RTS) needs to do to transport your data from one host to another, say as a message carrying the parameters of an entry method invocation. The RTS needs to find the size of each parameter it needs to package up, then (after allocating a buffer of the requisite size, with adequate space for the metadata, such as the entry method and chare to which the method invocation is addressed) traverse the data structure to actually copy the data into the outgoing buffer, and then, on the receiving side, to reconstruct each parameter by "unpacking" the data from the buffer into type-correct dsata structure. The PUP framework we describe below provides a general solution: instead of requiring programmers to provide sizing, packing, unpacking (and maybe other) specialized functions, it requires them to only define a PUP (for pack-unpack) method that specifies a traversal of each data type involved.

Let us start with a simple example, one that could be dealt with by the `PUPBytes` macro.

The class you want to pass as one of the parameters to an entry method is

This was the older : sectionPUP Framework.

Need to merge it with above/below

4.2 From The manual

The following description of PUP with examples is taken from the Charm++ manual. It should be mostly be retained with an ack saying its based on or taken from the manual. (So the publisher doesn't object to it later!).

The PUP (Pack/Unpack) framework is a generic way to describe the data in an object and to use that description for serialization. The Charm++ system can use this description to pack the object into a message and unpack the message into a new object on another processor.

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

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

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

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

4.3 PUP contract

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

Pup methods for complicated objects normally call the pup methods for their simpler parts. Since all objects depend on their immediate superclass, the first line of every pup method is a call to the superclass's pup method—the only time you shouldn't call your

superclass's pup method is when you don't have a superclass. If your superclass has no pup method, you must pup the values in the superclass yourself.

4.3.1 PUP operator

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

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

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

See `examples/charm++/PUP`

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

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

4.3.2 PUP STL Container Objects

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

See `examples/charm++/PUP/STLPUP`

4.3.3 PUP Dynamic Data

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

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

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

See `examples/charm++/PUP/HeapPUP`

4.3.4 PUP as bytes

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

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

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

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

4.3.5 PUP overhead

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

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

4.3.6 PUP modes

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

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

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

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

4.4 PUP Usage Sequence

Typical method invocation sequence of an object with a pup method is shown in Figure 4.1. As usual in C++, objects are constructed, do some processing, and are then destroyed.

Objects can be created in one of two ways: they can be created using a normal constructor as usual; or they can be created using their pup constructor. The pup constructor for Charm++ array elements and `PUP::able` objects is a "migration constructor" that takes a

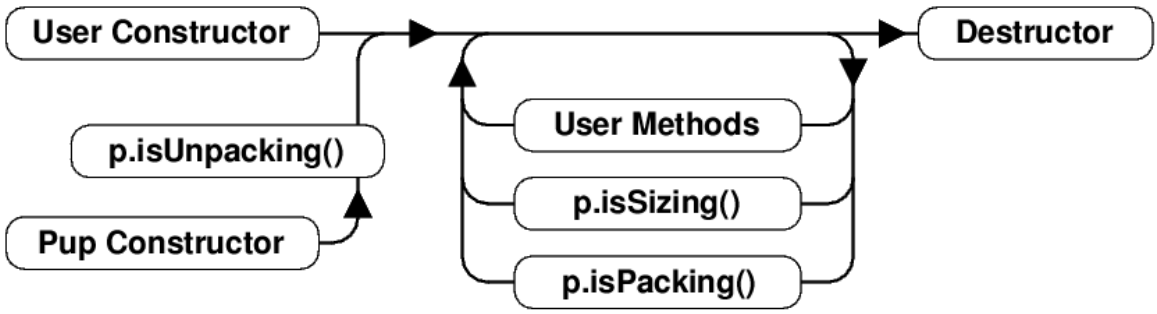

Figure 4.1: Method sequence of an object with a pup method.

single "CkMigrateMessage *"; for other objects, such as parameter marshalled objects, the pup constructor has no parameters. The pup constructor is always followed by a call to the object's pup method in `isUnpacking` mode.

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

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

4.5 Migratable Array Elements using PUP

Array objects can migrate from one PE to another. For example, the load balancer (see Section **??**) might migrate array elements to better balance the load between processors. For an array element to be migratable, it must implement a `pup` method. The standard PUP contract (see Section 4.3) and constraints wrt to serializing data apply.

A simple example for an array follows:

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

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

//In the .C file:
void A2::pup(PUP::er &p)
{
    p|nt;
    p|chr;
    p(flt,7);
    p|numDbl;
    if (p.isUnpacking()) dbl=new double[numDbl];
    p(dbl,numDbl);
}
```

The default assumption, as used in the example above, for the object state at PUP time is that a chare, and its member objects, could be migrated at any time while it is inactive, i.e. not executing an entry method. Actual migration time can be controlled (see section **??**) to be less frequent. If migration timing is fully user controlled, e.g., at the end of a synchronized load balancing step, then PUP implementation can be simplified to only transport "live" ephemeral data matching the object state which coincides with migration. More intricate state based PUPing, for objects whose memory footprint varies substantially with computation phase, can be handled by explicitly maintaining the object's phase in a member variable and implementing phase conditional logic in the PUP method (see Section **??**).

4.6 Marshalling User Defined Data Types via PUP

Parameter marshalling requires serialization and is therefore implemented using the PUP framework. User defined data types passed as parameters must abide by the standard PUP contract (see Section 4.3).

A simple example of using PUP to marshall user defined data types follows:

```
class Buffer {
public:
//...other declarations
  void pup(PUP::er &p) {
    // remember to pup your superclass if there is one
    p|size;
    if (p.isUnpacking())
```

```
      data = new int[size];
    PUParray(p, data, size);
  }

private:
  int size;
  int *data;
};


// In some .ci file
entry void process(Buffer &buf);
```

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

The order of marshalling operations on the send side is:

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

The order of marshalling operations on the receive side is:

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

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

See `examples/charm++/PUP/HeapPUP`

Chapter 5

Migration-based Load Balancing

A common scaling bottleneck for a wide range of applications is load imbalance. As applications are scaled to large systems, the probability of some cores receiving more work than others also increases. These over-loaded cores, which gradually lag behind, impact the progress of the entire execution negatively either due to communication dependencies of other cores on them, or simply because the tasks assigned to them are completed in a delayed manner. To this end, Charm++ provides support for dynamic load balancing enabled by its ability to place or migrate overdecomposed chares.

In our next example, we will demonstrate how load imbalance in an application leads to resource under utilization, following which we illustrate how the members of a chare array can be load balanced by the Charm++ runtime system. In this chapter, we will not cover the details of how the runtime system does the load balancing, i.e., how it decides which chares should be located on what processors, the ways in which it can interface with application code, etc. Instead, we will simply describe how do load balancing in Charm++ and show simple example codes that trigger load balancing using the AtSync method.

5.1 A Simple Synthetic Program to illustrate load-balancing

Our first example program consists of a single chare array created by the main chare. Each element of the Workers array has an entry method called nextStep, which executes a single computational step. In each step, it does a certain amount of work via the function doWork, and then proceeds in one of three ways:

- (a) if the desired number of iterations have been completed, it contributes to a reduction which calls CkExit and terminates the program
- (b) otherwise, if the current step is a multiple of 10, it invokes the load balancer by calling the AtSync() function provided by the runtime system

```
1  mainmodule loadBalancing {
2    mainchare Main {
3      entry Main(CkArgMsg *m);
4    };
5
6    array [1D] Workers {
7      entry Workers();
8      entry void nextStep();
9    };
10 };
```

Figure 5.1: load balancing: interface file

- (c) finally, if neither of the above are true, it proceeds to the next step by sending itself a nextStep method invocation

The AtSync function is the primary way for an application to coordinate load balancing with the runtime system. It is a way for the application to signal to the runtime system that the calling chare is at a natural synchronization point, and is ready to be migrated by the runtime system if need be. Once all chares call AtSync, the runtime system performs load balancing, migrating chares as it sees fit, and then calls ResumeFromSync on each chare to inform them that load balancing has completed and the computation can continue.

To use AtSync for load balancing, an application must do three things: set usesAtSync to true in each chares constructor, make sure the chares can be migrated by defining a pup method, and calling AtSync on every chare at an appropriate time. Figure 5.2 shows the code for the Workers chare array. In the constructor, usesAtSync is set to true so that the runtime system knows how many total AtSync calls it is waiting for. Chares that do not set usesAtSync should never call AtSync, but will still be migrated when load balancing occurs. The Workers chare array also defines a pup function which packs up its only member variable: stepNum. Finally, inside of the nextStep function, chares call AtSync on every tenth iteration.

When the runtime system detects that all elements of the Workers chare array have called AtSync, it collects load statistics and passes them to the current load balancing strategy. The strategy then decides how to remap the chares to processors to achieve a better balance of load. The runtime system carries out the necessary migrations to achieve this new mapping, and then calls ResumeFromSync on each element of the Workers array to inform them that they can continue working. In this case, they send themselves another nextStep message to continue the computation. Note that by calling AtSync, the application is informing the runtime system that the calling chare is at a synchronization point and is able to be migrated. Therefore, it is up to the application to ensure that any work done by the chare after the AtSync call is not dependent on the chares current location. It is generally advised that applications do not perform any work between the call to AtSync and when the runtime

```
25 class Workers: public CBase_Workers {
26   private:
27     int stepNum;
28
29   public:
30     Workers() : stepNum(0) {
31       usesAtSync = true;
32     }
33
34     Workers(CkMigrateMessage *m) { }
35
36     void pup(PUP::er &p) { p | stepNum; }
37
38     void nextStep() {
39       doWork(stepNum, thisIndex);
40       stepNum++;
41       if (stepNum == MAX_STEPS) {
42         contribute(CkCallback(CkCallback::ckExit));
43       } else if (stepNum % 10 == 0) {
44         AtSync();
45       } else {
46         thisProxy[thisIndex].nextStep();
47       }
48     }
49
50     void ResumeFromSync() { thisProxy[thisIndex].nextStep(); }
51 };
```

Figure 5.2: load balancing: chare array control logic

system calls ResumeFromSync. However, it should also be noted that the application should not rely on AtSync/ResumeFromSync as a synchronization mechanism. With more advanced load balancing techniques, the runtime system may not wait for all chares to call AtSync before it calls ResumeFromSync. This is also the case when there is no currently selected load balancing strategy.

To select a load balancing strategy for use, applications must link in the desired load balancing strategy and pass it in either at compile time or runtime. When compiling an application, to link against existing load balancing strategies, add -module ¡LBName¿ to your link command. There also exist two collections for linking large numbers of load balancers. Adding -module CommonLBs will link in a collection of most LBs present in the runtime system. Using -module EveryLB links in all load balancers, although some may have additional build dependencies when compiling Charm++. Once balancers are linked, you can add -balancer ¡LBName¿ to the compile line, or +balancer ¡LBName¿ to your run command to tell the runtime system to use that load balancing strategy when AtSync is called.

We will use the flexibility afforded by the doWork function to induce load imbalance in this computation. In a simple experiment, we let doWork function called by chare indexed i to execute for 100*i microseconds, irrespective of the stepNum. The resultant performance on ??? processors, with ??? chares, is visualized in a time-profile view provided by projections.

A little recap on how to obtain projections data and how to view the time profile.

For the first 10 steps, the execution is imbalanced. After that, the load balancing

5.2 PUP overview

SHOULD THIS MOVE BEFORE PREVIOUS SECTION? IN THE MIDDLE OF SEALIFE? AFTER SEALIFE TO ILLUSTRATE GENERAL CASE?

During load balancing, Charm++ RTS automatically decides on the new mapping of chares to the processors, and migrates them. In order to perform these migrations, the RTS needs to ensure that the chare being migrated retains its state, i.e. the data associated with a chare (user-allocated and system-allocated) is preserved. While the RTS can easily copy and maintain the system-allocated data, it needs assistance from the user to preserve user-allocated data. To this end, Charm++ RTS provides a PUP - pack and unpack - framework to the end user. User can use the serializer framework to handpick the data items that should be migrated with a chare, hence optimizing the communication by only migrating the necessary information.

The PUP framework is a serializer framework that can be used to inform the RTS of the actions (such as data allocation, copy to/from user memory etc) that should be taken to safely migrate a chare from one processor to another. Figure 5.3 presents the overview of the migration process using the PUP framework. A chare typically consists of stack-allocated and heap-allocated data members. The user writes an entry method pup for every migratable

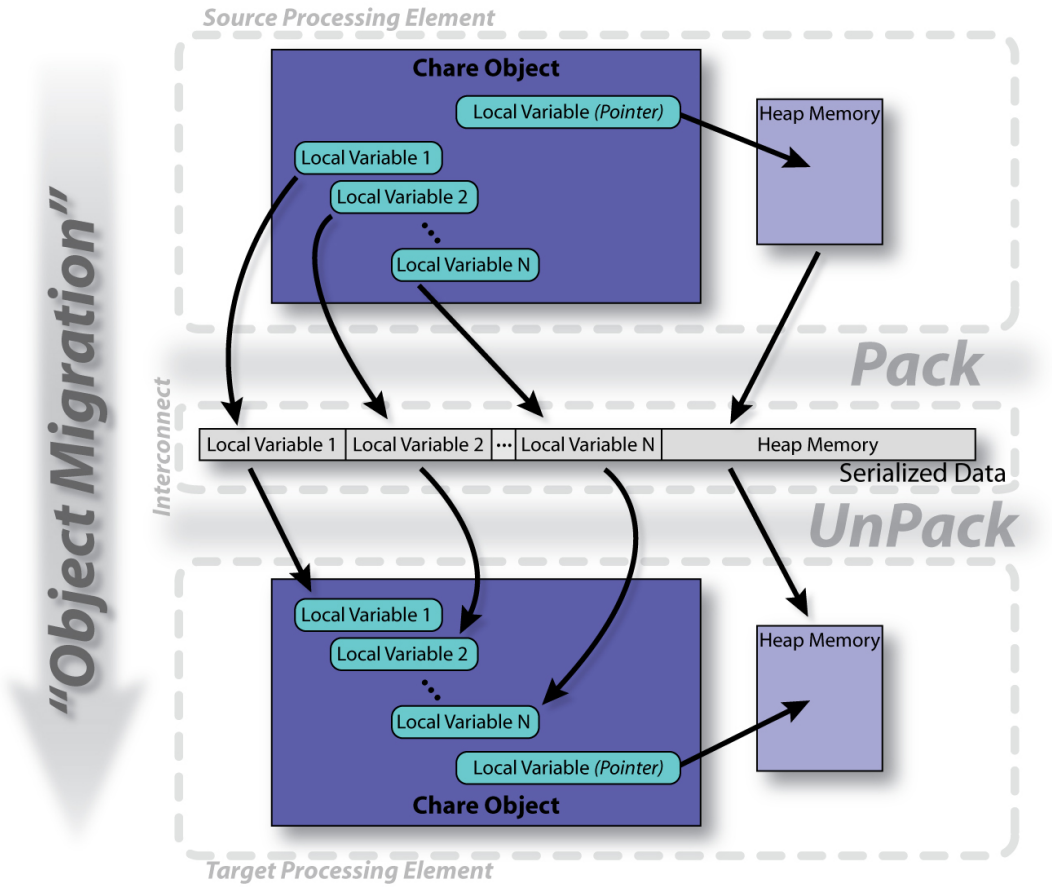

Figure 5.3: PUP process

```
1  class MyChare : public CBase_MyChare {
2    int a;
3    float b;
4    char c;
5    float localArray[LOCAL_SIZE];
6  };
```

```
1  void pup(PUP::er &p) {
2    CBase_MyChare::pup(p);
3    p | a;
4    p | b;
5    p | c;
6    p(localArray, LOCAL_SIZE);
7  }
```

Figure 5.4: Data members and pup method for a chare

```
1  class MyChare : public CBase_MyChare {
2    int heapArraySize;
3    float* heapArray;
4    MyClass *pointer;
5  };
```

```
1  void pup(PUP::er &p) {
2    CBase_MyChare::pup(p);
3    p | heapArraySize;
4    if (p.isUnpacking()) {
5      heapArray = new float[heapArraySize];
6    }
7    p(heapArray, heapArraySize);
8    bool isNull = !pointer;
9    p | isNull;
10   if (!isNull) {
11     if (p.isUnpacking()) pointer = new
           MyClass();
12     p | *pointer;
13   }
14 }
```

Figure 5.5: PUPing memory allocated on heap

chare class, which tells the RTS about the data members that are allocated on heap/stack, and copied from/to the user memory to/from the serialized buffer. Using this method, a chare is serialized into a long byte-array, and sent to the destination. At the destination, the same pup method is used, in the reverse direction, to recreate the chare.

Figure 5.4 shows the pup method for a chare with data members of basic types (int, float etc), and an array of fixed size. The argument to the method is an object of the system defined class PUP::er. Based on when the pup method is invoked (during packing, or unpacking etc), this object performs different tasks. The first statement in the pup method is a call to the pup method of the base class. This call ensures that any data member inherited from the base class are packed/unpacked properly. The pipeline operator (|) using the PUP::er class is overloaded to perform the necessary operations - copy data to serialzied buffer during packing and copy data to user memory during unpacking. To pack/unpack an array, the length of the array should be specified.

In Figure 5.5, we present an example for packing/unpacking heap allocated memory. Unlike stack allocated memory, which is assigned as an object is created, heap allocated memory need to be explicitly allocated by the user. The PUP::er class provides a query function isUnpacking that can be used to determine the stage of migration process during which the pup method is invoked. The important thing to remember is that during unpacking, any memory that should be allocated on heap should be allocated before the data is copied to it. For example, as shown in Figure 5.5 (line 5), the heapArray is allocated first, and then the packing/unpacking method is called on it (line 7). Similar order has to be imposed on objects of classes that are created using the new operator (lines 11-12).

5.3 SeaLife

As our second example in this chapter, we use a simple program to simulate life within the sea. Consider a three dimensional space with a regularly spaced grid embedded into it. Let each point in the grid contain one of four objects - a fish, a shark, water or a coral stone (Figure 5.6). The task is to simulate activity in this environment in an iterative manner for a number of time steps following the given rules:

In the description below, we use the word "neighbors" of a cell to mean those cells immediately adjacent to it, i.e. only one of whose coordinates differs from it, and by a magnitude of 1. (See Figure 5.7) for a 2-dimensional illustration. There are 4 neighbors in 2 dimensions and 6 in 3D. In contrast, the phrase "neighborhood" refers to adjacent cells for whom every coordinate can differ by at most 1 from its. There are 9 cells in the neighborhood in 2D and 27 in 3D, including itself.

- Boundary elements are always stones, except for the top layer, where all elements are water.
- The locations with stone will always have stone.
- If a location is currently occupied by water, a fish will occupy it in the next time step if more than half of its immediate neighbors are fishes; if more than half of its neighbors are sharks, a shark will occupy it in the next time step (consider the 7 point stencil).
- If a location is currently occupied by a shark, the shark will die if there is no fish in its neighborhood (consider the 27 points stencil), and the location will be occupied by water in the next step.
- If a location is currently occupied by a fish, the fish will die if there is no fish in its neighborhood, or if more than half objects around it are sharks (consider 27 point stencil); the location will be occupied by water if the fish dies.

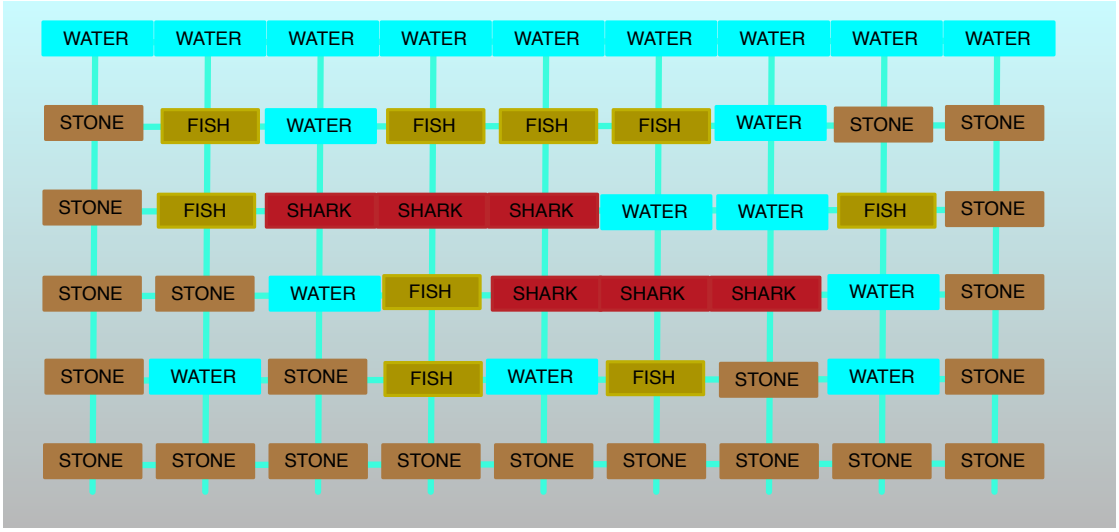

Figure 5.6: SeaLife - simulation set up.

- The initial distribution of objects is random, with some regions containing more stones/water, while the rest contain more fishes/sharks.

In each time step, for every grid point, based on the object placed at it and its surrounding locations, the object for the next iteration is decided.

The parallel construction of this simulation is similar to the stencil example discussed earlier. Upon execution, the mainchare creates a one dimensional chare array of type SeaLife. The 3D-grid is divided among chares of this chare array using one dimensional domain decomposition. Figure 5.8 presents the chare array definition with focus on its data members. The SeaLife class primarily consist of information about grid points owned by it and some other variables for bookkeeping purpose.

Having created the chare array, the mainchare invokes entry method run on the array. The SDAG function run controls the execution of each chare for rest of the simulation (Figure 5.9). At the beginning of every iteration, each chare sends the information about objects residing on its either boundary to its neighbor. We assume a wrap around of the 3-D grid to keep the code simple. After sending the information, a chare waits to receive information from its neighbor (line 27). Having received the information, work is performed and the simulation moves onto the next iteration (line 30).

Finally, we present the work function that progresses the simulation by updating the type of object that will occupy the grid points. The most important thing to note in Figure 5.10 is the variation in amount of computation performed based on type of object that reside at

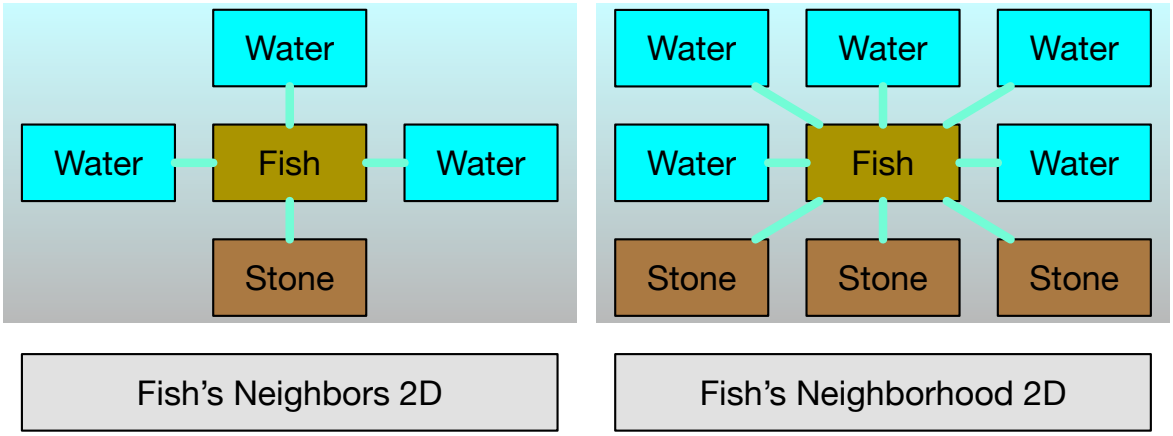

Figure 5.7: Neighbors - Neighborhood

```
47 class SeaLife: public CBase_SeaLife {
48   SeaLife_SDAG_CODE
49
50   public:
51     int iter, msg;
52     int ***objectType, ***altobjectType;
```

Figure 5.8: SeaLife: data members

```
18    entry void run() {
19      for(iter = 0; iter < numiters; iter++) {
20        serial "send" {
21          if(thisIndex == 0 && iter % 1 == 0) {
22            CkPrintf("[%d] Starting iter %d on chare %d\n",CkMyPe(),iter,thisIndex);
23          }
24          sendNeighbors();
25        }
26
27        for(msg = 0; msg < 2; msg++) {
28          when recvNeighbor[iter](int ref, int dir, int msgLength, int vals[msgLength]) serial {
29            processMsg(dir, msgLength, vals);
30          }
31        }
32
33        serial "do_work" { work(); }
34      }
35      serial { contribute(CkCallback(CkReductionTarget(Main,done),mainProxy)); }
36    };
```

Figure 5.9: SeaLife: control code in SDAG

a grid point. If the object at a grid point is stone, least amount of work is needed. For water, one needs to read 7 grid points before the update happens, whereas for the rest, 27 grid points need to be accessed.

Let us now run this simulation. We execute this code on a 8-core desktop for a system with 64×32 grid points per chare. We create a chare array with 80 elements. This execution for 150 time steps completes in 25s. Figure 5.11 presents the *time line* view of the above run using Projections that reveal the load imbalance among the processors. Processor 6 is heavily over loaded, while processors $1-2$ perform very little useful work. Other processors, such as $3-5$, also spend significant amount of time idling. Many processors finish their execution much before the overloaded processor 6, and idle wait for it to finish. Let us explore if enabling load balancing for this problem, as described in the previous section, helps reduce the idle time.

5.3.1 Load balancing SeaLife

We begin our task, of enabling load balancing, by defining the pup method for class SeaLife. The code is shown in Figure 5.12, which is also added to the C++ file. The first line of the method is a call to the pup method of the base class, and is followed by invocation of __sdag_pup. Thereafter, the members of basic data types (iter, msg) are pupped using the

```
173     void work() {
174       for(int i = 1; i < length+1; i++) {
175         for(int j = 1; j < spaceDim+1; j++) {
176           for(int k = 1; k < spaceDim+1; k++) {
177             if(objectType[i][j][k] == STONE) {
178               altobjectType[i][j][k] = STONE;
179             } else if(objectType[i][j][k] == WATER) {
180               int nums[4];
181               nums[0] = nums[1] = nums[2] = nums[3] = 0;
182               for(int diff = −1; diff <= 1; diff += 2) {
183                 nums[objectType[i+diff][j][k]]++;
184                 nums[objectType[i][j+diff][k]]++;
185                 nums[objectType[i][j][k+diff]]++;
186               }
187               if(nums[0] > 4) altobjectType[i][j][k] = FISH;
188               else if(nums[1] > 4) altobjectType[i][j][k] = SHARK;
189               else altobjectType[i][j][k] = WATER;
190             } else {
191               int nums[4];
192               nums[0] = nums[1] = nums[2] = nums[3] = 0;
193               for(int diffi = −1; diffi <= 1; diffi += 1)
194                 for(int diffj = −1; diffj <= 1; diffj += 1)
195                   for(int diffk = −1; diffk <= 1; diffk += 1)
196                     nums[objectType[i+diffi][j+diffj][k+diffk]]++;
197               altobjectType[i][j][k] = objectType[i][j][k];
198               if(objectType[i][j][k] == FISH) {
199                 if(nums[FISH] == 1) altobjectType[i][j][k] = WATER;
200                 if(nums[SHARK] >= 13 ) altobjectType[i][j][k] = WATER;
201               } else {
202                 if(nums[FISH] == 0) altobjectType[i][j][k] = WATER;
203               }
204             }
205           }
206         }
207       }
208       int ***tempVal;
209       tempVal = objectType;
210       objectType = altobjectType;
211       altobjectType = tempVal;
212     }
```

Figure 5.10: SeaLife: differential work based on object type

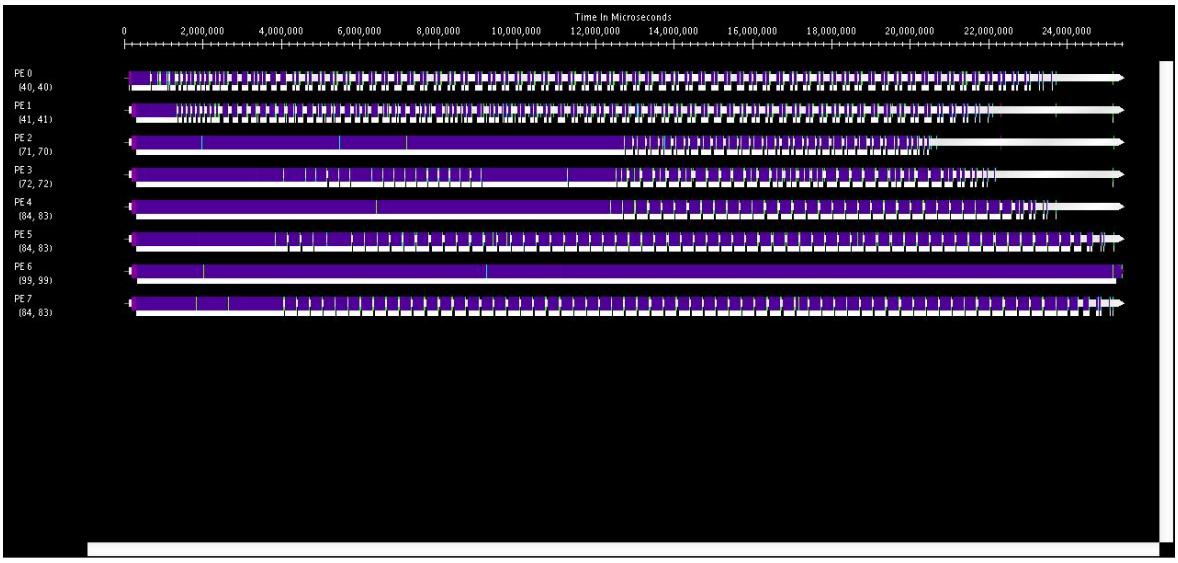

Figure 5.11: SeaLife - timeline for base run.

predefined pipeline operator. Finally, to pack/unpack the data array, we first need to ensure that the array is properly allocated when the chare is being unpacked. Notice the call to forceBoundary method, which is invoked to enforce boundary conditions. This call is needed because to reduce communication cost, we chose not to pack/unpack the data that is enforced by the boundary condition. Hence, the pack/unpack call is only made on the active data (lines 236-240).

With the correct pup method in place, we add the periodic invocation of AtSync to the control flow in run entry method as shown in Figure 5.13. After the work has been performed for an iteration, every LDB_PERIOD time steps, each chare invokes AtSync.

Having enabled the load balancing framework, the execution time dropped to 20s from 25s – a 20% performance gain. More importantly, a time line view of this run using Projections (Figure 5.14) clearly shows that once load balancing is performed, the load on all cores is very similar, and the idle time goes down significantly. In real world scenarios, when a more complex version of this code is executed for thousands of time steps, the improvement will be higher.

```
215     void pup(PUP::er &p)
216     {
217       CBase_SeaLife::pup(p);
218       __sdag_pup(p);
219       p|iter;
220       p|msg;
221
222       if (p.isUnpacking()) {
223         objectType = new int**[length+2];
224         altobjectType = new int**[length+2];
225         for(int i = 0; i < length+2; i++) {
226           objectType[i] = new int*[spaceDim+2];
227           altobjectType[i] = new int*[spaceDim+2];
228           for(int j = 0; j < spaceDim+2; j++) {
229             objectType[i][j] = new int[spaceDim+2];
230             altobjectType[i][j] = new int[spaceDim+2];
231           }
232         }
233         forceBoundary();
234       }
235
236       for(int i = 1; i < length+1; i++) {
237         for(int j = 1; j < spaceDim+1; j++) {
238           p(objectType[i][j], spaceDim+1);
239         }
240       }
241     }
```

Figure 5.12: SeaLife: pup method in C++ file

```
34        serial "do_work" { work(); }
35
36        if(iter && (iter % LDB_PERIOD == 0)) {
37          serial { AtSync(); }
38          when ResumeFromSync() { }
39        }
40      }
41      serial { contribute(CkCallback(CkReductionTarget(Main,done),mainProxy)); }
```

Figure 5.13: SeaLife: adding call to AtSync in SDAG

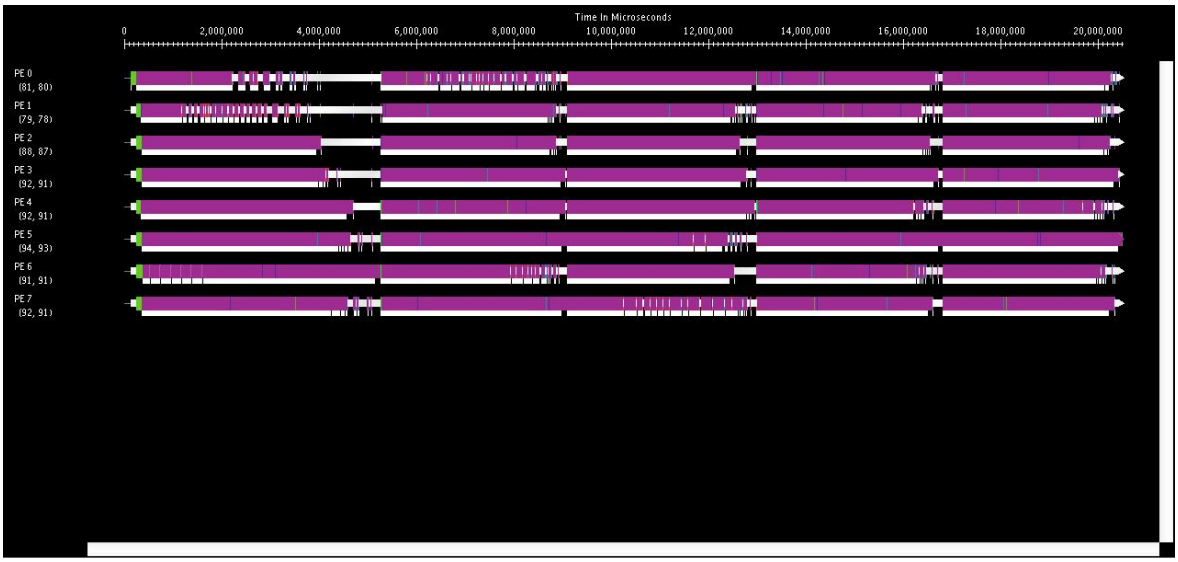

Figure 5.14: SeaLife - time for load balanced run.

Chapter 6

Checkpointing

This chapter presents *checkpointing*, a fundamental mechanism that permits the runtime system to store the state of an application. You will learn in which situations this feature comes handy and what capabilities Charm++ provides to dump a program's data to either memory or disk. The checkpointing framework in Charm++ leverages the PUP mechanism presented in Chapter 4.2.

6.1 Saving the State of an Application

Charm++ permits an application to serialize its data for migration purposes. This is achieved through the PUP mechanism that is flexible enough to allow a user selectively marshal only the necessary data. Checkpointing is based on the same PUP framework to be able to save the state of an application.

Checkpointing is all about saving the state of an application. There are several situations in which it is useful to save the state of an application:

- Control the progress of an application. Consider, for instance, a long running program, which has an execution time that exceeds the time limit of a single job submission. For this application to make progress, it is necessary to run the application for a time lapse within the limit of a job submission and store its state before the end of the time limit. Later on, on another job submission the application can be resumed from the saved state and be able to finish with execution.

- Reuse results in modular execution. For applications that run in phases, where each phase consists in the execution of a module of the application, checkpointing may provide a way to reuse results of one phase to the next. Figure 6.1 presents a scenario where an application runs a module in the first phase (denoted by $Module_1$). The output of this phase can be used by several different modules in the second phase (named

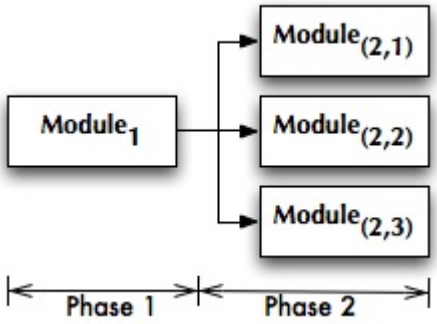

Figure 6.1: Use of checkpointing to separately run phases of an application.

$Module_{(2,i)}$). If modules in the second phase are run in different job submissions, then checkpointing may allow the reuse of the output of one phase to be used by several modules in the next phase.

- Provide fault tolerance. When failures are frequent relative to the execution time of an application, we may use checkpointing to obtain a backup of the application to be used in case a failure strikes the system. A popular mechanism consists in taking checkpoints periodically and resuming from the last checkpoint after a failure occurs.

The state of an application may be saved in different devices, provided that there is enough room for all the data. A checkpoint may be stored in a local disk or in the NFS disk. It may also be stored in main memory or distributely in the memory of other nodes. New technologies, like flash drives or solid state disks, will increase the number of possible recipients of a checkpoint. Charm++ permits checkpoints to be stored in either disk or main memory. Disk checkpointing can be applied towards any of the general situations described above. Memory checkpoint, on the other hand, only makes sense for fault tolerance and will be examined in the next subsection.

The standard Charm++ build comes with the disk checkpointing mechanism built in. In order to checkpoint an application, the main chare must call the following function:

```
void CkStartCheckpoint(char* dirname,const CkCallback& cb);
```

However, notice this function is meant to be called from only the main chare. It is not a collective operation and instead requires the entire system to synchronize before making this call. For example, a global reduction to the main chare is a good place to locate this call. Let us consider a more concrete case. Recall the 2D 5-point stencil program from Chapter 3. In that code, each tile compute the value of all its entries based on the values of neighboring

tiles. After each iteration, all tiles participate in a reduction to determine if another iteration is necessary. Once the reduction completes at the main chare, we may trigger the checkpoint call depending on how much progress we have made since the last checkpoint.

In order to make the 2D 5-point stencil program able to checkpoint, we will need to make a few changes to the code. We will guide you through each of these changes.

First of all, we will use the number of iterations completed as a guide on when a checkpoint should be triggered. A constant value `CKPT_FREQ` will determine the number of iterations between checkpoints. Then, we will need to make variable `numStepsCompleted` a class member, to be able to store its value in the checkpoint. Figure 6.2 summarizes these two additions to the code.

```
7  #define CKPT_FREQ 10
```

```
17 class Main : public CBase_Main {
18  private:
19   int numStepsCompleted;
```

Figure 6.2: Changes in file main.h.

Next, and probably the most important change to the code is to add PUP methods. We already saw in Chapter 4.2 how to serialize data of a chare. The same mechanism is used for checkpointing. Figure 6.3 presents the two PUP methods required for each class, `Main` and `Tile`, respectively. In the case of `Main`, we only care for one variable, which stands for the number of iterations completed. On the other hand, `Tile` has a more complicated PUP method. We start by computing the size of the tile by multiplying the respective coordinates of the tile. If the PUP method is unpacking we need to reserve space for both `tileData` and `scratchData`. Nevertheless, when we store the data contained in `Tile` class, we only store `tileData` since it is the only data we need to recover to make progress in the application. We must highlight that temporary data is not part of the checkpoint.

Being able to select what to checkpoint and what not is one of the advantages of *application-level* checkpoint. This scheme requires the involvement of the user to tell the runtime system what portion of the data is required in the checkpoint, saving only those data structures that are strictly necessary. This is opposed to *system-level* checkpoint, where the runtime system would take the whole process and serialize it, including temporary variables and values in registers.

Since checkpoint is only performed at those points in the program specified by the programmer, then PUP methods store all necessary information to resume the program from the checkpoint. Charm++ may be instructed to have *any time migration*, where a chare can

```
99    CBase_Main::pup(p);
100   p | numStepsCompleted;
101 }
102
```

```
168 void Tile::pup(PUP::er &p){
169   int i,tileSize = (tileWidth + 2) * (tileHeight + 2);
170   CBase_Tile::pup(p);
171   p | eventCounter;
172   if(p.isUnpacking()){
173     tileData = new float[tileSize];
174     scratchData = new float[tileSize];
175   }
176   for(i=0; i<tileSize; i++){
177     p | tileData[i];
178   }
179 }
```

Figure 6.3: Pup methods for Main and Tile classes.

be migrated at any point in time, not necessarily when chares synchronize. This feature is totally compatible with checkpointing.

Another important change in the code is related to actual function call for checkpointing an application. Figure 6.4 presents the actual code in `Main` class that will check whether it is time to start a checkpoint phase or not. Notice that the first parameter specifies the directory where the checkpoint is to be stored. The second parameter is the callback to be made once the checkpoint is complete. This callback has the same effect as the case where there is no checkpoint.

```
35 void Main::reductionCallback(float data) {
36   // Get the result of the reduction
37   float maxStepDiff = data;
38
39   // Display this step's maximum difference to the user
40   CkPrintf("Step %d: %f\n", ++numStepsCompleted, maxStepDiff);
41
42   // Check to see if the difference is small enough that the
43   //   application can complete
44   if (maxStepDiff <= targetDiff) {
45     CkExit();          // Program finished
46   } else {
47     if(numStepsCompleted % CKPT_FREQ == 0){
48       CkCallback cb(CkIndex_Tile::startStep(),grid);
49       CkStartCheckpoint("ckpt",cb);
50     } else {
51       grid.startStep();  // Another step needs to be run
52     }
53   }
54 }
55
```

Figure 6.4: Code to start checkpointing.

Finally, to complete the list of changes of the original code, we need to make simple but fundamental changes. First of all, variable `grid` has to be declared as readonly. Secondly, class `Main` must be declared as `migratable`. A summary of these changes appear in figure 6.5.

Running the application with disk checkpointing does not require any changes to the command line:

```
8   readonly CProxy_Tile grid;

10  mainchare [migratable] Main {
```

Figure 6.5: Last changes to make checkpointing able to work.

```
./charmrun +p4 ./jacobi 0.01 10 10 10 10
```

Figure 6.6 presents the output of the execution of the program that checkpoints every `CKPT_FREQ` iterations.

In order to restart an execution from a checkpoint, we must use `+restart` option in the command line and provide the directory where the checkpoint was stored. Notice that we may restart using a different number of processors:

```
./charmrun +p2 ./jacobi 0.01 10 10 10 10 +restart ckpt/
```

Figure 6.7 offers the output of restart. Only the iterations after the last checkpoint are executed.

6.2 Fault Tolerance

Supercomputers comprise a large number of processing elements and many more other components. As a complex system, supercomputers are not trivial to maintain. It is difficult to prevent all possible scenarios in which one or a set of those components will fail. For instance, alpha particules may hit circuits and provoke bit flips in memory. Some memory errors can be corrected with the error correction codes (ECC) in the memory module, while others are just detected but uncorrectable.

Each part of a machine has a probability of failure that can be translated into *mean time to failure* (MTTF) or, in other words, how much time on average the component will run before breaking. Given the MTTF of each component, we can compute the MTTF of a system composed by N components using the formula:

$$MTTF_{System} = \frac{1}{\sum_{i=1}^{N} \frac{1}{MTTF_i}}$$

It is not hard to see that the more components, the smaller the MTTF of the whole system and thus failures will be more frequent. Imagine a supercomputer with millions of

```
Converse/Charm++ Commit ID: v6.3.0-372-g9429fbd
Charm++> scheduler running in netpoll mode.
Charm++> Running on 1 unique compute nodes (2-way SMP).
Charm++> cpu topology info is gathered in 0.041 seconds.
"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
[0] Checkpoint starting in ckpt
Checkpoint to disk finished in 0.252618s, sending out the cb...
Step 11: 0.010295
Step 12: 0.009353
```

Figure 6.6: Output of code with disk checkpointing.

```
Converse/Charm++ Commit ID: v6.3.0-372-g9429fbd
Charm++> scheduler running in netpoll mode.
Charm++> Running on 1 unique compute nodes (2-way SMP).
Charm++> cpu topology info is gathered in 0.000 seconds.
[0]CkRestartMain done. sending out callback.
Step 11: 0.010295
Step 12: 0.009353
```

Figure 6.7: Output of code when restarting with disk checkpointing.

cores, hundreds of thousands of memory modules, thousands of disks and hundreds of routers. It would not surprise anybody if such machine experienced a failure every hour.

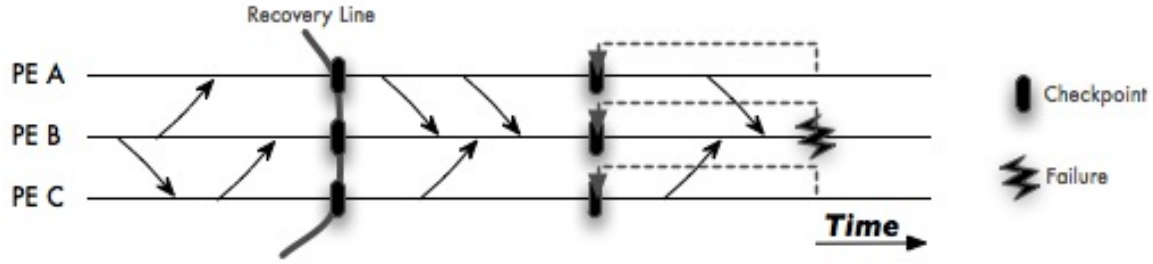

Figure 6.8: Use of checkpointing to provide fault tolerance.

One way to handle the problem of multiple failures in current supercomputers is by using the checkpoint framework and building what is known as a *rollback-recovery* mechanism. Figure 6.8 presents an example of how rollback-recovery works. Suppose there are 3 processing elements, named A, B and C. Objects living on those PEs will exchange messages during normal execution until they all reach a checkpoint call. The collection of the checkpoint of all PEs is called *recovery line* and constitutes the point of the execution where all PEs will be rolled back once we find a failure. Since failures are not totally predictable, checkpoints will be periodically taken. The higher MTTF of a system, the more frequent the checkpoints.

If a one of the PEs fail, then the system will make the whole collection of PEs to roll back to the most recent checkpoint and restart from there. Although checkpoints may be stored on NFS disk and be retrieved by any PE, Charm++ also provides an in-memory checkpoint mechanism. In this case, each PE will checkpoint to the memory of other PE (called the *buddy*) and to its own memory as well. If one PE fails, the buddy PE will provide the checkpoint for the failed PE.

This in-memory checkpointing scheme works similar to disk checkpoint mechanism. The following function is used to start the checkpoint:

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

Using the same example as in the previous subsection, we only need to modify the function call at class `Main` to look like figure 6.9. Notice that Charm++ will automatically react to the failure and restart the PEs from their respective checkpoints. The programmer has to make the necessary changes in the code and determine an appropriate checkpoint frequency. Having frequent checkpoint will reduce the amount of work that has to be *redone* after a failure. However, every time a checkpoint is performed, there is a performance penaliza-

tion. Therefore, the checkpoint period is really a tradeoff between having a low overhead for checkpointing and reducing the amount of work to do again in case of a failure.

```
35 void Main::reductionCallback(float data) {
36   // Get the result of the reduction
37   float maxStepDiff = data;
38
39   // Display this step's maximum difference to the user
40   CkPrintf("Step %d: %f\n", ++numStepsCompleted, maxStepDiff);
41
42   // Check to see if the difference is small enough that the
43   //   application can complete
44   if (maxStepDiff <= targetDiff) {
45     CkExit();          // Program finished
46   } else {
47     if(numStepsCompleted % CKPT_FREQ == 0){
48       CkCallback cb(CkIndex_Tile::startStep(),grid);
49       CkStartMemCheckpoint(cb);
50     } else {
51       grid.startStep();  // Another step needs to be run
52     }
53   }
54 }
55
```

Figure 6.9: Code to start in-memory checkpointing.

In order to have in-memory checkpoint available, Charm++ has to be built using the `syncft` flag, for instance:

```
./build charm++ net-linux-x86_64 syncft
```

Chapter 7

Structured Dagger (SDAG)

Consider the recursive Fibonacci program described in Section 2.9.3 for calculating the n-th Fibonacci number. We can describe the behavior of the chare that computes fib(n) as follows (Presumably, for the exercise 2.9.3, your solution followed the same pattern) :

1. When the constructor is called, if n is small enough, calculate fib(n) sequentially and return the result to the parent. Otherwise, fire two children chares, one to calculate fib($n-1$) and the other to calculate fib($n-2$).

2. If the response method is invoked, count down the number of responses remaining, and add the contribution to a running total. If no more responses remain, send the result to the parent or print the result if this chare is the parent.

This *reactive* specification is valid. It captures what the chare *can* do. But as programmers we know more than that. For instance, if a chare C fires two children, each child *will* send a response to C. So, we can specify the actions of the chare's more definitively:

1. When the constructor is called, do the same as the above.

2. When the two children respond (and they will), add up their contributions and send the result to the parent.

This difference, though subtle in the small example at hand, is important nonetheless. The latter is a more accurate statement of the chare's function and describes its life cycle clearly. The SDAG notation is designed to express the latter view, without losing efficiency. The increase in clarity of expression is even more evident when we consider more complex programs.

Figure 7.1 shows the SDAG code for calculating the n-th Fibonacci number. The definitions of sequential C++ functions such as seqFib and respond are shown in Figure 7.2. At

```
1  mainmodule fib {
2    mainchare Main {
3      entry Main(CkArgMsg* m);
4    };
5
6    chare Fib {
7      entry Fib(int n, bool isRoot, CProxy_Fib parent);
8      entry void calc(int n) {
9        if (n < THRESHOLD) serial { respond(seqFib(n)); }
10       else {
11         serial {
12           CProxy_Fib::ckNew(n − 1, false, thisProxy);
13           CProxy_Fib::ckNew(n − 2, false, thisProxy);
14         }
15         when response(int val)
16           when response(int val2)
17             serial { respond(val + val2); }
18       }
19     };
20     entry void response(int);
21   };
22 };
```

Figure 7.1: SDAG code for the example calculating Fibonacci numbers.

this point, the main thing to notice about SDAG code in Charm++ is that it is added to the ci file, and is translated to C++ code using charmc. We advise the reader to revisit the code for computing n^{th} Fibonacci number in Figure 7.1, once we have describe various SDAG keywords and their utility.

7.1 Specifying Dependencies in SDAG: the when Clause

The reactive specification of Charm++ programs makes it clear what actions are performed by a chare when a particular kind of entry method is invoked on it. However, if there are dependencies between these entry methods within a single chare, they are specified implicitly in the code. Moreover, since Charm++ does not guarantee any order on the delivery of messages, these dependencies must be specified at the target of the invocations. These dependencies are usually enforced through variables that track the state of the target chare, causing messages to be buffered until they are ready to be processed by it.

In the case of the Fibonacci example, the dependencies are tracked without SDAG by counting the number of responses from the children. Using SDAG, we eliminate this manual

```
1  #include "fib.decl.h"
2  #define THRESHOLD 10
3
4  class Main : public CBase_Main {
5  public:
6    Main(CkArgMsg* m) { CProxy_Fib::ckNew(atoi(m->argv[1]), true, CProxy_Fib()); }
7  };
8
9  class Fib : public CBase_Fib {
10 public:
11   Fib_SDAG_CODE
12   CProxy_Fib parent; bool isRoot;
13
14   Fib(int n, bool isRoot_, CProxy_Fib parent_)
15     : parent(parent_), isRoot(isRoot_) {
16     calc(n);
17   }
18
19   int seqFib(int n) { return (n < 2) ? n : seqFib(n - 1) + seqFib(n - 2); }
20
21   void respond(int val) {
22     if (!isRoot) {
23       parent.response(val);
24       delete this;
25     } else {
26       CkPrintf("Fibonacci number is: %d\n", val);
27       CkExit();
28     }
29   }
30 };
31
32 #include "fib.def.h"
```

Figure 7.2: C++ code for the example calculating Fibonacci numbers.

dependency tracking by specifying two nested when triggers that cause the SDAG sequence to block until the when trigger is satisfied.

In general, SDAG specifies a sequence of execution. So in the following snippet, statement1 will be executed before statement2:

```
1  <statement1>
2  <statement2>
```

If statement1 is a when clause that waits on a dependency, the statement2 will not execute until the dependency is fulfilled, and the corresponding block is executed. Seen this way, a when clause can be thought of as a blocking receive statement.

SDAG code is written in the .ci file because it is parsed by the Charm++ interface translator. Instead of just specifying the interface for an entry method in the .ci file (as we have done for all entry methods so far), an SDAG method is completely written in the .ci file.

We can specify a dependency (or a when clase) in the following way:

```
1  chare foo {
2    entry method2(paramenters) {
3      when method1(paramenters) { /* block1 */ }
4      <statement2>
5    };
6    entry method1(paramenters);
7  };
```

In the above snippet, when method2 is invoked, SDAG waits for an invocation of method1 before continuing execution. If this invocation has already arrived, then block1 will execute. If not, the SDAG scheduler will enqueue a continuation contingent on method1 arriving. Once method1 arrives, block1 will execute, followed by statement2. Note that method1 must be declared as a entry method, even if it is used in a when clause.

To wait for multiple dependencies, we can do the following:

```
1  when method1(paramenters), method2(paramenters) { /* block1 */ }
2  <statement2>
```

In this case, block1 will not execute until both method1 and method2 arrive. The previous code snippet is slightly different from the following. The above when clause is only fulfilled when both methods arrive, but the following is fulfilled incrementally (first method1 arrives, then the code waits for method2). Once method1 arrives the runtime is allowed to execute down this path.

```
1  when method1(paramenters)
2    when method2(paramenters) { /* block1 */ }
3  <statement2>
```

In addition, if we have a block of code to execute when method1 arrives and a different

block when method2 arrives, and method2 is dependent on method1, we can just sequence the two when statements. The different in this case is that the set of parameters to method1 will not be available to method2. Hence, the memory usage will be lower, because those parameters will be freed by the underlying runtime system.

```
1  when method1(paramenters) { /* block1 */ }
2  when method2(paramenters) { /* block2 */ }
3  <statement2>
```

7.2 Blocks of C++ Code: the serial Clause

Due to the details of how the Charm++ interface file is parsed, C++ blocks of code must be marked by the serial construct. (The keyword atomic, which is deprecated, is an alias for serial.) Hence, if block1 is a C++ block of code that we want to sequentially execute, we could do the following:

```
1  when method1(paramenters) serial { /* block1: C++ block of code */ }
2  when method2(paramenters) { /* block2 */ }
3  <statement2>
```

In this case, block1 can be an arbitrary block of C++ code that accesses member variables of the encapsulating chare.

7.3 Other SDAG Constructs

SDAG has several other constructs that have similar semantics to their sequential C++ counterparts. SDAG supports an if-then-else block:

```
1  when method1(parameters) /* block1 */
2  if (<condition>)
3    when method2(parameters) /* block2 */
4  else
5    when method3(parameters) /* block3 */
```

In the above code snippet, depending on the runtime evaluation of the condition, either method2 or method3 will be waited on by SDAG.

Now that a few of the basic SDAG constructs have been introduced, let's reexamine the Fibonacci example. On line 16 of Figure 7.2, the calc SDAG method is called in the constructor, which will start the sequence of SDAG code. Note that a SDAG method does not need to be called remotely: it can be called as a local inline method invocation.

In the calc method, we first check if n $<$ THRESHOLD, and then respond to the parent if we are below the parallel calculation threshold. If not, we execute a C++ block that

creates the two children chares (lines 12-13). Once we create the two children, we wait for two responses by specifying two nested when clauses. Each when clause names the incoming parameter val differently, so they do not collide. Once both responses arrive, we call the respond method with the augmentation of these two values (line 17).

7.3.1 Using SDAG in a Charm++ Program

The other change to the Fibonacci program is on line 11 of Figure 7.2. Because the Fib chare includes SDAG code, we must insert a special macro Fib_SDAG_CODE that includes the code that is generated by the Charm++ interface translator. In general, if a chare (or mainchare or array) Foo includes SDAG code, we must insert a macro Foo_SDAG_CODE in the C++ class declaration for that chare.

7.3.2 The for Clause

SDAG also has a for clause that allows a block to be executed n times in sequence:

```
1  for (i = 0; i < n; i++) /* block1 */
```

The above snippet will execute block1 n times. Just like a sequential program, block1 will be able to access i, but note that i must be declared as a class member variable. This restriction is because a SDAG for loop is not executed sequentially, and hence there is no stack to store the i variable.

We can wait on a dependency n times, in the following way:

```
1  for (i = 0; i < n; i++)
2    when method1(paramenters) /* block1 */
```

In this snippet, before moving on the next statement, SDAG will wait for method1 to arrive n times and execute block1 each time method1 arrives. Note that this code does not specify an ordering on method1. So for instance, if some other chare has a proxy to this chare, it can make the following invocations:

```
1  serial {
2    for (int i = 0; i < n; i++)
3      proxy.method1(paramenters)
4  }
```

With this pattern, SDAG will wait for all method1 to arrive n times, but these n invocations will not be ordered. If ordering is required, we could create n different entry methods and wait for them in order. However, this would be very verbose and does not allow n to be dynamic. Instead, *reference numbers* can be used to *tag* a entry method invocation. Then, syntax can be used to wait on a certain tag of particular method. If the first parameter of an

SDAG method is an integer, this implicitly becomes that method's tag. So the above example could be modified in the following way:

```
1  serial {
2    for (int i = 0; i < n; i++)
3      proxy.method1(i, paramenters)
4  }
```

In this case, the n invocations of method1 will be tagged with the value of variable i as they are sent. We can then use the following syntax in SDAG to wait on each dependency in order:

```
1  for (i = 0; i < n; i++)
2    when method1[i](int ref, paramenters) /* block1 */
```

The added code [i], causes each when clause to only be satisfied when method1 arrives tagged with i (i.e. ref == i). In general, we can specify a certain reference number of any when clause:

```
1  when method1[<reference–number>](int ref, paramenters) /* block1 */
```

The above snippet will wait for method1 with the reference number reference-number (i.e. ref == reference-number). Note that ref is not a special variable name, but a method that is tagged must include an integer as the first parameter (and that implicitly becomes its reference number). The reference-number can be a integer variable that holds the tag, an integer value known at compile-time, or an expression that evaluates to an integer. For example, if we want to wait on reference number 100:

```
1  when method1[100](int ref, paramenters) /* block1 */
```

7.4 Example: Parallel Prefix Sum

An important algorithm used for many parallel operations is the calculation of a prefix sum. A prefix sum over an array of length n, can be expressed sequentially as follows:

```
1  for (int i = 1; i < n; i++)
2    A[i] += A[i–1];
```

After the execution of the above loop, each element i in array A will contain the sum of all the previous elements' values.

Consider how we might parallelize this sequential algorithm. The problem with simply parallelizing the sequential loop shown above is that each index of the loop is dependent on the previous value, which will effectively sequentialize the execution, even if we distribute array A over a set of a parallel entities.

The key to parallelizing this algorithm is examining the actual dependencies more closely.

Note that to obtain the final value for $A[i]$, we just need to sum all the values $A[0]+\cdots+A[i-1]$. The sequential version of this algorithm eliminates extra work by incrementally computing the sum, completing the operation in n steps. To effectively parallelize this algorithm, we must increase the amount of work, but the gained parallelism will hopefully cause execution to be faster, even though we are executing more instructions overall.

We can describe a slightly different algorithm that executes $\lg n$ steps, each step summing up to $n - \lg i$ values from neighboring elements. In the following snippet, during each step each element augments the values dist away from it.

```
1 for (int dist = 1; dist < n; dist <<= 1)
2   for (int i = n − 1; i >= dist; i−−)
3     A[i] += A[i − dist];
```

When the algorithm has been expressed in this manner, we can parallelize the algorithm by creating a n-dimensional chare array, where each element holds one value. Each element exchanges the necessary information for each step and then moves on to the next.

Figures 7.3 and 7.4 present an SDAG program applying the prefix sum operation on an array of integer elements. PrefixSum is a one-dimensional chare-array that performs the prefix-sum operation. Each chare array element holds a single integer, stored in the member variable myVal. The start entry method specifies the actions that each chare takes and the dependencies between them. The SDAG for loop on line 13 uses a loop index variable dist, which is doubled at the end of each iteration, varying from 1 to N, the number of elements in the chare array. On line 14 the if construct that distinguishes between chares that forward their running sums to their dist-away neighbors, from those that do not. On line 17 we specify a dependency on recv if that chare was the target of an invocation on line 15. Once a chare has completed $\lg N$ number of iterations, it exits the for loop and prints out its final value in the prefix sum (line 21) and makes its contribution to the reduction to the mainchare on line 22.

Note that for each recv we are tagging it with thisIndex + dist. Without a tag, since each chare may receive multiple recv invocations from different iterations, these may be mixed up causing an incorrectly result. By tagging the invocation, we ensure that the recv invocations are processed in the appropiate order for each chare without using a barrier or reduction between them to synchronize between iterations.

7.5 Enhancing Concurrency

The when clause in SDAG imposes strict/rigid ordering among the actions a chare can execute. This may restricts concurrency within an object compared with what one can do with plain Charm++. The following constructs can be used to explicitly overlap sequences of SDAG code to relax ordering constraints by explicitly specifying which sequences can be executed in parallel.

```
1  mainmodule prefixSum {
2    readonly CProxy_Main mainProxy;
3
4    mainchare Main {
5      entry Main(CkArgMsg *m);
6      entry [reductiontarget] void done();
7    };
8
9    array [1D] PrefixSum {
10     entry PrefixSum(int length);
11     entry void recv(int ref, int val);
12     entry void start() {
13       for (dist = 1; dist < N; dist <<= 1){
14         if (thisIndex + dist < N) serial {
15           thisProxy[thisIndex + dist].recv(dist, myVal);
16         }
17         if (thisIndex >= dist)
18           when recv[dist](int ref, int val) serial { myVal += val; }
19       }
20       serial {
21         CkPrintf("[%d] value %d\n", thisIndex, myVal);
22         contribute(CkCallback(CkReductionTarget(Main, done),mainProxy));
23       }
24     };
25   };
26 };
```

Figure 7.3: The .ci file for a simple program that calculates a parallel prefix sum using SDAG.

```
1  #include "prefixSum.decl.h"
2
3  /* readonly */ CProxy_Main mainProxy;
4
5  class PrefixSum : public CBase_PrefixSum {
6    int N, myVal, dist;
7  public:
8    PrefixSum_SDAG_CODE
9
10   PrefixSum(int length)
11     : N(length)
12     , myVal(1) { }
13
14   PrefixSum(CkMigrateMessage *m) { }
15 };
16
17 class Main : public CBase_Main {
18 public:
19   Main(CkArgMsg *m){
20     mainProxy = thisProxy;
21     CkAssert(m->argc == 2);
22     int length = atoi(m->argv[1]);
23
24     CProxy_PrefixSum proxy = CProxy_PrefixSum::ckNew(length,length);
25     proxy.start();
26   }
27
28   void done() { CkExit(); }
29 };
30
31 #include "prefixSum.def.h"
```

Figure 7.4: The corresponding C++ code.

7.5.1 The forall Clause

If we have a set of sequenced operations that can happen in any order, we can use the forall clause to specify n of them:

```
1  forall [i] (0:10,1) /* block1 */
```

In the above snippet, block1 is overlapped: that is, in this case block1 is executed 10 times without any sequencing between those 10 invocations. Now, if block1 is a serial block that contains C++ code, this simply specifies that block1 can be executed in any order. If block1 contains when clauses, the ith iteration of the forall does not wait for the iteration $i-1$ of the forall. In this way, the iterations of the forall are overlapped.

In general, a forall is specified as following:

```
1  forall [<member−variable>] (<min>:<max>,<stride>) /* block1 */
```

The range between min and max is inclusive.

We can still use reference numbers in a forall to specify that we are waiting on a possibly-strided range of reference numbers, but the corresponding blocks can execute in any order:

```
1  forall [i] (0:n,1)
2    when method1[i](int ref, parameters) /* block1 */
```

As an example for using the forall statement in SDAG, we show a snippet of code from computational fluid dynamics, which is called the Harlow-Welch scheme. For this method, we decompose the 3D space into rectangular blocks from the X and Y dimension. Each iteration consists of exchanging the boundary elements with the four neighbors in the 2-D grid. This computation is performed concurrently for each plane in the Z dimension, and each plane converges independently. We could represent this pattern in the following snippet of SDAG code:

```
1  entry void Harlow_Welch() {
2    serial {
3      initialize();
4      for (int i = 0; i < NZ; i++)
5        hasConverged[i] = false;
6    }
7    forall [i] (0:NZ−1,1) {
8      while (!hasConverged[i]) {
9        // copy and send messages
10       when recvN[i](BdryMsg *n), recvS[i](BdryMsg *s), recvE[i](BdryMsg *e), recvW[i](BdryMsg *w)
11         serial {
12           int my_conv = update(i, n, s, e, w);
13           CkCallback cb(CkReductionTarget(Tile, checkConverged), thisProxy);
14           cb.setRefnum(i);
15           contribute(sizeof(int), &my_conv, CkReduction::logical_and, cb);
```

```
16          }
17        when checkConverged[i](int converged) serial {
18          hasConverged[i] = converged;
19        }
20      }
21    }
22  }
```

7.5.2 The overlap Clause

The forall clause is useful when we have n sequences to overlap that are similar. If the sequences are different, we can use the overlap clause:

```
1  overlap {
2    <statement1>
3    <statement2>
4  }
```

In the above snippet, statement1 will be overlapped with the execution of statement2. SDAG will not proceed past the overlap statement, until all statements contained within are fully executed.

For example. if three entry methods can be received in any order, we could write the following:

```
1  overlap {
2    when method1(parameter) /* block1 */
3    when method2(parameter) /* block2 */
4    when method3(parameter) /* block3 */
5  }
```

In this case, this code is equivalent to having three regular Charm++ entry methods with sequential C++ code in block1, block2, and block3.

7.6 More Constructs

7.6.1 The while Clause

In addition to a for clause, SDAG also allows while clauses that have identical syntax and semantics as a sequential C++ while statement:

```
1  while (<condition>) /* block1 */
```

In the above snippet, block1 is executed until condition is evaluated as false. This is a parallel while loop so a when inside of block1 may cause the scheduler to wait for the dependency to

be satisfied before continuing.

7.6.2 The case Clause

SDAG also allows a disjunction to be specified between a set of when clauses. In other words, the case statement is satisfied when one of the specified when clauses is satisfied:

```
1  case {
2    when method1(parameters) /* block1 */
3    when method2(parameters) /* block2 */
4    when method3(parameters), method4(parameters) /* block3 */
5  }
6  <statement2>
```

In this case, if method1 arrives, block1 will execute and then SDAG will move on to statement2. The same is true if method2 arrives, except that block2 is executed. If both method3 and method4 arrive, block3 will execute and then statement2. Note that the when clause must be fully satisfied for that "branch" to be satisfied. Also, if a branch of the case clause is partially satisfied, that method reception can be used later.

7.7 Example: 5-point Stencil

Remember in Section 3.5.1 we had to use a counter to count the number of expected events and we used a reduction every iteration to synchronize the computation. By using SDAG, we can eliminate the error-prone manual counters and use reference numbers to tag invocations per iteration thereby eliminating the need for explicit synchronization each iteration. Instead, each iteration we can perform a asynchronous reduction that is overlapped with several iterations, so the computation does not wait for all the elements to synchronize before the next iteration continues.

Figure 7.5 shows the updated code for the Tile chare array using SDAG and asynchronous reductions to determine convergence every 5 iterations.

Each Tile element has a while loop that continues until the converged boolean is set to true. While this is true, we send the ghost regions and then wait on all 4 ghost regions to arrive. An overlap is used here because the ghost regions can arrive in any order and we want to update our corresponding region as soon as any region arrives. After all of those arrive, we run the doCalc kernel, which now returns the max difference its found locally during the update. Every 5 iterations we contribute to an asynchrouous reduction that logically ANDs whether a Tile's local computed difference is below the target convergence criteria. The logical AND ensures that if any of the Tile's are above the threshold we will collectively not reach convergence. 4 iterations later, we wait on the result from the asynchrouous reduction and then send the callback to the mainchare if we have reached convergence so the program

```
1  module tile {
2    array [2D] Tile {
3      entry Tile();
4      entry void startStep(CkCallback finished) {
5        while (!converged) {
6          serial { sendGhostRegions(); }
7          overlap {
8            when recvNorthGhost[iter](int ref, float ghostData[dataLen], int dataLen) serial {
9              memcpy(tileData + NORTH_OFFSET, ghostData, dataLen * sizeof(float));
10           }
11           when recvSouthGhost[iter](int ref, float ghostData[dataLen], int dataLen) serial {
12             memcpy(tileData + SOUTH_OFFSET, ghostData, dataLen * sizeof(float));
13           }
14           when recvWestGhost[iter](int ref, float ghostData[dataLen], int dataLen) serial {
15             for (int i = 0; i < tileHeight; i++)
16               tileData[WEST_OFFSET + (TILE_Y_STEP * i)] = ghostData[i];
17           }
18           when recvEastGhost[iter](int ref, float ghostData[dataLen], int dataLen) serial {
19             for (int i = 0; i < tileHeight; i++)
20               tileData[EAST_OFFSET + (TILE_Y_STEP * i)] = ghostData[i];
21           }
22         }
23         serial {
24           int c = doCalc() < targetDiff;
25           CkCallback cb(CkReductionTarget(Tile, checkConverged), thisProxy);
26           if (iter % 5 == 1) contribute(sizeof(int), &c, CkReduction::logical_and, cb);
27         }
28         if (++iter % 5 == 0) {
29           when checkConverged(bool result)
30             if (result) serial {
31               if (thisIndex.x == 0 && thisIndex.y == 0) {
32                 CkPrintf("converged in %d iterations\n", iter);
33                 finished.send();
34               }
35               converged = true;
36             }
37         }
38       }
39     };
40     entry [reductiontarget] void checkConverged(bool result);
41     entry void recvNorthGhost(int ref, float ghostData[dataLen], int dataLen);
42     entry void recvSouthGhost(int ref, float ghostData[dataLen], int dataLen);
43     entry void recvWestGhost(int ref, float ghostData[dataLen], int dataLen);
44     entry void recvEastGhost(int ref, float ghostData[dataLen], int dataLen);
45   };
46 }
```

Figure 7.5: The .ci file for the Tile chare array for the 5-point stencil.

can exit and set the converged variable to true.

SOme old comments: Traffic sim using SDAG: no barrier

7.8 Exercises

Prefix sum?

Chapter 8

Threaded Entry Methods

Imagine a situation in which you are writing the code for an entry method of a chare, and realize that you need to obtain a value from another chare. What you can do, using techniques we learned in earlier chapters, is to send a request to that chare, and have it send a response to one of your entry methods, where you continue your execution. This technique, sometmes called "continuation passing" style, works, but makes programs look awkward, especially in the simple scenarios, such as the request-response scenario described above.

In this chapter, we will first learn threaded entry methods, and their counterpart the blocking or "sync" entry methods. Then we see examples of how blocking invocations of sync methods are carried out from threaded entry methods, and how to utilize this mechanism for expressing control flow within an object effectively. We next learn about *"future"*, a useful construct when using threaded entry methods. Finally, we will learn a general (and therefore, low-level) way explicitly suspending and awakening threaded entry methods.

8.1 Threaded and Sync Methods

One can declare an entry method to be "threaded" simply by adding the attribute "threaded" to the entry method declaration in the interface file:

```
entry [threaded] void f(...);
```

This tells the charm runtime system to create a thread every time this method is invoked, and run it within its own thread. An object may have multiple threaded entry methods, and at any particular time there may be many instances of threaded entry methods in operation; However, in practice, the common usage is to have a single thread associated with each object.

A method can be declared to be a `sync` method using the attribute "sync" in its declaration in the interface file.

```
entry [sync] MsgType * g(..);
```

A sync method must have a return type which is a pointer to a message type (see Section 9.2). This is in contrast to normal entry methods which must have a void as a return type.

```
1  mainmodule checkOrder {
2    message MsgData;
3    readonly int numElements;
4
5    mainchare Main {
6      entry Main(CkArgMsg *msg);
7      entry [reductiontarget] void isSorted(int result);
8    };
9
10   array [1D] SimpleArray {
11     entry SimpleArray();
12     entry [threaded] void run();
13     entry [sync] MsgData * getValue();
14   };
15 }
```

Figure 8.1: A Blocking invocation from a threaded entry method : the interface file `blocking.ci`

8.2 A simple illustrative example of blocking invocation

Let us consider the following problem: we have a chare-array A of N elements, and each element holds a single value. We want to check if the values are already in a sorted order: i.e. value held by i^{th} element of the chare array is less than or equal to value held by j^{th} element iff $i < j$. One way is to have each chare with an index i (except the last one), compare it's own value with that held by the chare next to it (i.e. with an index $i + 1$). We could make a chain of these checks, so A[0] sends its value to A[1], which checks it and if it is in order sends its own value to A[2] and so on. If the last chare (index $N - 1$) finds everything in order, it sends a reply to the main chare reporting success. If any chare finds that its value is out of order with the previous chare's value, it reports a failure to the main chare, and does not send a message to its next chare. (Exercise: write this program).

But this is not a very efficient way of checking. It takes a *sequence* of N messages to get the result back to main chare, and only one chare is doing the check at a time, i.e., there is no parallelism in this way of doing things.

Instead, we can have each chare check with its right neighbor, in parallel, and have them combine there results via a reduction (which takes only $logP$ time, with P processors). We will do this using a threaded entry method, just to illustrate how to use them. The program

```
1  #include "checkOrder.decl.h"
2  #include <stdlib.h>
3
4  /* readonly */ int numElements;
5
6  class MsgData: public CMessage_MsgData
7  { public:   double value; };
8
9  class Main : public CBase_Main {
10  public:
11   Main(CkMigrateMessage *msg) { }
12   Main(CkArgMsg* msg) {
13     numElements = 10;
14     if (msg->argc > 1) numElements = atoi(msg->argv[1]);
15     delete msg;
16
17     // Create the chare array with numElements and set callBack
18     CProxy_SimpleArray A = CProxy_SimpleArray::ckNew(numElements);
19     CkCallback *cb = new CkCallback(CkReductionTarget(Main, isSorted), thisProxy);
20     A.ckSetReductionClient(cb);
21     A.run(); // start the run thread of each object.
22   }
23
24   void isSorted(int result) {
25     if(result == 1) {
26       CkPrintf("The array was sorted \n");
27     } else {
28       CkPrintf("The array was not sorted \n");
29     }
30     CkExit();
31    }
32 };
```

Figure 8.2: A Blocking invocation from a threaded entry method: class Main in the C++ file `blocking.C`

is shown in Figures 8.1, 8.2 and 8.3. A threaded entry method is declared by adding the keyword `threaded` in its declaration in the `.ci` file, as shown in line 12 of Figure 8.1. A threaded method is allowed to suspend or make blocking calls. The simplest blocking call we will learn is the invocation of a blocking (or "sync") method. Here, method `getValue` is declared as a sync method, in the `.ci` file. `sync` methods return a pointer to a message, unlike the regular entry methods, which have a void return value.

Class Main is shown in Figure 8.2. This class performs simple tasks - creates chare-array `A`, sets reduction callback to one of its method and invokes `run` on `A`. When invoked,

```
34 class SimpleArray : public CBase_SimpleArray {
35  private:
36   double myValue;
37
38  public:
39   SimpleArray(CkMigrateMessage *msg) { }
40   SimpleArray() { myValue = drand48();  }
41
42   void run() {
43     int result = 1;
44     printf("[%d](%d): myValue = %f\n", thisIndex, CkMyPe(), myValue);
45     if(thisIndex != numElements-1) {
46       MsgData *m = thisProxy(thisIndex+1).getValue();
47       if(myValue > m->value) result = 0;
48     }
49     contribute(sizeof(int), &result, CkReduction::logical_and);
50   }
51
52   MsgData* getValue() {
53     MsgData * m = new MsgData();
54     m->value = myValue;
55     return m;
56   }
57 };
58
59 #include "checkOrder.def.h"
```

Figure 8.3: A Blocking Invocation from a threaded entry method: class SimpleArray in the C++ file `blocking.C`

the constructor of chare-array `A` initializes the value every chare contains using a pseudo-random value (Figure 8.3). Thereafter, when `run` method is invoked, it calls `getValue`, a `sync` method, on the chare to its right in `A`, and obtains the value stored in it. Having obtained the value, it compares that value with its own value, and stores the result. Finally, it contributes the result to the reduction, whose result is delivered to the main chare.

The code of the `run` method clearly expresses what the chare is doing. Of course, for this simple example, you could implement it without the threaded method, with simple code. (Exercise: write this program!). But you can see how the threaded/sync methods work, and can now imagine more complex scenarios where they will be useful.

.

8.3 Futures in Threaded Entry Methods

Charm++ supports a simplified idea of the well-known "future" construct. In Charm++ *Future* is a construct that you can create from a threaded entry method; it will contain a value at some future time. If you "touch" a Future before it has a value, the calling thread blocks (i.e. suspends) "touching" here simply means accessing the value, and you have to use a special function call to access the value.

The function calls provided by Charm++ for implementing its version of *futures* are as follows:

- `CkFuture` is a system defined type that holds a "future" value.
- `CkFuture CkCreateFuture()` creates and returns a future structure. This is an opaque structure (i.e. you don't get to see its data members or what else is inside of it, except via the function calls described here) that can be sent in messages to other chares.
- `CkWaitFuture(fut)` function, given a future structure (`fut`) as a parameter, checks if the future value has been set; if so it returns with the value (which must be a pointer to a `message` type). If not, it suspends the thread and resumes it when the value is available.
- `CkSendToFuture(f, m)` A remote chare which holds a future structure can set its value by calling: `CkSendToFuture(f, m)` where f is a future and m is a `message *` (i.e. pointer to a message).
- You can also "probe" to check if a future's value is set, without making the thread block, using `CkProbeFuture()`...but we do not use that call in our example below.

8.3.1 Fibonacci Numbers with Futures

As a simple example, we will use recursive definition of n^{th} Fibonacci number:

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

(Again, not the best way of computing n^{th} Fibonacci number, but this formulation is a conventional stand-in for all divide-and-conquer algorithms.)

We already saw how this was computed using ordinary entry methods in an earlier chapter (**REF**). Here, let us see if expressing it using threaded methods can simplify its expression.

The idea for the program using *futures* is: we will make the constructor call "run", a threaded entry method; When we create a child chare, we will provide it with a reference to a newly created future (one future for each call). Then we just wait for each of them in succession by trying to access their values. Once both futures have been "realized" (i.e. have their values set), we add their values, and set the future that our parent sent us, using `CkSendToFuture`.

```
1  mainmodule fib {
2    message ValueMsg;
3
4    mainchare Main {
5      entry Main(CkArgMsg *m);
6      entry [threaded] void run(int n);
7    };
8
9    chare Fib {
10     entry Fib(int n, CkFuture f);
11     entry [threaded] void run(int n, CkFuture f);
12   };
13 };
```

Figure 8.4: Finding fibonnaci using Future: the interface file `fib_futures.ci`

Figures 8.4 and 8.5 show the code corresponding to this idea.

Since the constructor of a chare is not allowed to be a threaded entry method, we use a threaded `run` method in the main chare as well as the `Fib` chare. Whenever we create a child chare, we create a future first (`f, f1, f2`) and pass it as a constructor parameter to it. We wait for the futures to be set (lines 39 and 40) and then compute a result. The result is filled into a newly created message, and sent to the parent's future (line 46).

Incidentally, threaded methods don't *have* to be named `run`; It is just a convention to do so, especially when there is a single threaded method associated with a chare, and it captures the overall life-cycle of the chare.

8.3.2 Explitly suspending and awakening threads

A more general method for controlling threads is to explicitly suspend the currently running thread via `CthSuspend()` call, and to `awaken` it using `CthAwaken` call. To use this method, you need to obtain your own thread "handle" via the `CthSelf()` call.

Fibonacci using explicit control

Why couldn't we use `sync` methods for the fibonacci example? The problem is that we need to fire two instances of the fib chare. If we were to use sync method for that, we would be stuck waiting for the result of the first cal, before we can make the second call. That would serialize the computation. As another way of implementing doubly recursive Fibonacci with threads, we will use "bare" threads in this example:

The lowest level primitives you are allowed to call on a thread are:

```
13 class Main : public CBase_Main {
14 public:
15   Main(CkMigrateMessage *m) {};
16   Main(CkArgMsg* m) { thisProxy.run(atoi(m->argv[1])); }
17   void run(int n) {
18     CkFuture f = CkCreateFuture();
19     CProxy_Fib::ckNew(n, f);
20     ValueMsg *m = (ValueMsg*)CkWaitFuture(f);
21     CkPrintf("The requested Fibonacci number is : %d\n", m->value);
22     CkExit();
23   }
24 };
25
26 class Fib : public CBase_Fib {
27 public:
28   int result;
29   Fib(CkMigrateMessage *m) {};
30   Fib(int n, CkFuture f){ thisProxy.run(n, f); }
31
32   void run(int n, CkFuture f) {
33     if (n < THRESHOLD) result = seqFib(n);
34     else {
35       CkFuture f1 = CkCreateFuture();
36       CkFuture f2 = CkCreateFuture();
37       CProxy_Fib::ckNew(n-1,  f1);
38       CProxy_Fib::ckNew(n-2,  f2);
39       ValueMsg* m1 = (ValueMsg*)CkWaitFuture(f1);
40       ValueMsg* m2 = (ValueMsg*)CkWaitFuture(f2);
41       result = m1->value + m2->value;
42       delete m1; delete m2;
43     }
44     ValueMsg *m = new ValueMsg();
45     m->value = result;
46     CkSendToFuture(f, m);
47   }
48 };
49 #include "fib.def.h"
```

Figure 8.5: Finding fibonnaci using Future: the C++ file `fib_futures.C`

- `CthSuspend()` to suspend the currently running thread, and return control to the scheduler;

- `CthAwaken(threadID)` to put the thread with a given threadID (presumably suspended at the time of the call) in the scheduler's queue of ready threads.

- In addition, a function CthSelf() returns the threadID of the running thread.

```
1  mainmodule fib_threads {
2
3    mainchare Main {
4      entry Main(CkArgMsg *m);
5    };
6
7    chare fib {
8      entry fib(int amIroot, int n, CProxy_fib parent);
9      entry [threaded] void run(int n);
10     entry void response(int);
11   };
12 };
```

Figure 8.6: Finding fibonnaci using threads: the interface file `fib_threads.ci`

So, our idea is this: we will invoke a threaded entry method called "run" from the constructor of the Fibonacci chare. It will fire two children chares, which will return result to our "response" entry method . However, after firing both children, the calling entry method simply suspends itself, after storing its threadID in an object variable, `tid`. The "response" method adds up the results being returned, counts up the number of response received, and when it has received two responses, "awakens" the suspended thread. Note that `CthAwaken(tid)` only puts the thread `tid` in the ready queue, and does not immediately return control to it. The thread must wait for its turn, since there may be other chares and messages waiting to be executed on this processor. When scheduled, the `run` thread simply continues (i.e. it is as if it "returned" from the CthSuspend call), and sends a response to its parent.

We need to take into account who the parent is: if your parent is another fib chare (as will be the case for most chares), you send the result to it's response method. But if you are the root of the tree of chares, created by the main chare, then you just print the result and terminate the program by calling CkExit().

```
9  class Main : public CBase_Main
10 {
11   public:
12     Main(CkMigrateMessage *m) {}
13     Main(CkArgMsg * m) {
14       if(m->argc < 2) CmiAbort("./fib_threads N");
15       int n = atoi(m->argv[1]);
16       CProxy_fib pfib;
17       CProxy_fib::ckNew(1, n, pfib);
18     }
19 };
20
21 class fib : public CBase_fib
22 {
23   private:
24     int result, count, IamRoot;
25     CthThread  tid; CProxy_fib parent;
26   public:
27     fib(CkMigrateMessage *m) {}
28     fib(int amIRoot, int n, CProxy_fib _parent) {
29       IamRoot = amIRoot;
30       parent = _parent;
31       thisProxy.run(n);
32     }
33
34     void run(int n) {
35       tid = CthSelf();
36       if (n< THRESHOLD) {
37         result = seqFib(n);
38       } else {
39         CProxy_fib::ckNew(0,n-1, thisProxy);
40         CProxy_fib::ckNew(0,n-2, thisProxy);
41         result = 0;
42         count = 2;
43         CthSuspend(); }
44       if (IamRoot) {
45         CkPrintf("The requested Fibonacci number is : %d\n", result);
46         CkExit();
47       } else parent.response(result);
48     }
49
50     void response(int fibValue) {
51       result += fibValue;
52       count--;
53       if(!count)        CthAwaken(tid);
54     }
55 };
56 #include "fib_threads.def.h"
```

Figure 8.7: Finding fibonnaci using threads: the C++ file `fib_threads.C`

Figures 8.6 and 8.7 show the code corresponding to this idea. Make sure you understand how it expresses each of the ideas mentioned above.

How does this code compare with the original code which did not use threaded entry methods? Not that much simpler or shorter, is it? But at least now you know the most elementary method for controlling threads. And yes, the `run` thread kind of looks nicer, since it captures the logic in one place; except that we have to understand how the response method captures the data and resumes the thread.

I think a simple example, where the root uses a serial-looking CG, and all the parallel operations are hidden in broadcasts to do mat-vec mupltiplies, and reductions, will be good to show here. Either just show the top level (main chare or the "controller" chare) code, or also show full code using a simple denese-matrix version.

8.4 Callback for resuming a thread

We saw the callbacks in an earlier section 3.3.1 in the context of reductions. Callbacks allow a general purpose way of transfering control. They are often used in a library to tranfer control back to the calling (client) module.

Charm++ supports a special callback for threaded entry methods which provides a neat method to resume a thread after some operation is completed. This callback is called `CkCallbackResumeThread`.

From a threaded entry method, you can invoke some operation in (say) another chare or a broadcast to an entry method in an entire chare array, passing to it this callback, making sure you construct it as a parameter.

For example:

```
ArrayProxy[i].doWork(..., CkCallbackResumeThread(), ..);
```

This has the nornal intended effect of sending the freshly created callback object as a parameter to doWork. In addition, it ends up suspending the calling thread. * [1] The called entry method (or another object to which the callback has been forwarded) can resume this thread by calling `send()` method on the passed callback object, like so:

```
cb.send()
```

This will end up sending a message to the chare of suspended entry method, resuming the thread!

You can also send a message as a parameter to the `send()` call, as explained below. With this usage, you can achieve the functionality of a`sync` method, without using a sync method. This is a more general usage, because `sync` methods requir you to generate a return message

[1] For the curious: this happens because the destructor of the callback includes a `cthSuspend()` call. The destructor is naturally called when the invocation is sent to the target object (`ArrayProxy`).

right away (at the end of sync method). In contrast, with this mechanism, the called method can pass the callback to another chare, which can pass it to another, and so on, and then the last one can send a reply back to the suspended thread.

For this usage, the calling (threaded) method does the following:

```
// Inside a threaded entry method
....
MyMessage *m;

ArrayProxy[i].doWork(..., CkCallbackResumeThread((void \*\&) m), .. );
//when the thread is resumed, the ‘‘result’’ is available in the the
message pointed to by m .
```

A common usage pattern for using `CkCallbackResumeThread()` is to use the callback in a reduction. You broadcast to an EP of a chare array, supplying a callback as before, and the array returns the result via a reduction, using the provided callback for the reduction results.

8.4.1 Finding Median

As a example to illustrate the use of `CkCallbackResumeThread`, consider the problem of finding the median of a large number of floating point numbers spread across a elements of a chare array. Let us say that we decided to do this using an iterative process that refines a guessed median repeatedly. The main chare will create the chare array (called `Partition` in our example) that holds the numbers, and the constructor of each element of Partition will create a set of random numbers (say, in the range 0.0 to 1.0). The main chare will make an initial guess for the median, and broadcast to Partition. Each element of Partition, when it gets the query from the main chare, calculates how many of the numbers it holds are smaller than the suggested median, and how many are larger, and contributes those two counts into a reduction. The main chare, when it receives the global sum of those counts, adjusts its guess for the median accordingly, and broadcasts it to the array again. The cycle repeats, until the 2 counts are sufficiently close (I.e. almost equal. More concretely, the difference between the counts is less than 1% of the total count of numbers stored).

This algorithm is implemented using threaded entry method as shown in the program of Figures 8.8, 8.9 and 8.10. Note the use of a CkReductionMsg being passed to `CkCallbackResumeThread`. Since the results are coming back in the form of the result of a reduction, this special message type is necessary. You can extract the data fileds corresponding to the reduction results by accessing the `data` field of the message.

Of course, you could implement this algorithm using structured dagger as well, or even just plain Charm++. But now imagine that the median finding is needed as a part of some broader computation. The main chare's run method may have called some other method `f`

```
1  mainmodule Median {
2
3    mainchare Main {
4      entry Main(CkArgMsg* m);
5      entry [threaded] void computeMedian();
6    };
7
8    array [1D] Partition {
9      entry Partition(void);
10     entry void queryCounts(double median, CkCallback &cb);
11   };
12 };
```

Figure 8.8: Finding median using threads: the interface file `median.ci`

which calls `g`, etc. and somewhere deep in the call chain, you want to call the median finding function (which is not an entry method, but a regular private method or a stand-alone function). It will be somewhat difficult to write that code using `sdag`.

8.5 When not to use threads

Threaded methods can be convenient, and threads are pretty light weight. For example, creation and context switching between threads typically cost less than a microsecond on today's machines. But they do add to the cost that tiny bit; in contrast, the structured dagger has a much lower overhead.

More significantly, a thread requires a stack of its own. By default, the system creates a pretty small stack (typically 4 KB). You can specify a larger stack for a threaded entry method using the following command line argument: `+stacksize <size-in-bytes>`.

Yet, the problem is the need to specify adequate stack size makes the method somewhat error prone, and possibly wasteful, because a stack once allocated occupies memory until the threaded method finishes, even when the stack is not "filled".

Migrating a chare when it contains a suspended threaded method is challenging, the but the runtime supports this. It needs to use somewhat expensive system calls (`mmap`) to do this. Also, since it has to migrate the entire stack (it is error-prone to migrate only the "used" portion of the stack), it adds somewhat to the migration cost.

These are some reasons why you should avoid threads and use structured dagger as much as possible. But there are situations where a threaded method is indispensable. For example, if an entry method `M` calls a function `f` which calls a function/method `g`, and inside of `g`, you need some value or data form another chare. It is much more convenient to make `M` a threaded method, and use a sync method call inside `g` to get the remote value. Without `M`

being threaded, you will have to split the code of `M`, `f` and `g`, and put the parts that execute after the remote value is returned into separate methods or functions. Such situations, along with situations when elegance is more important than small performance impact (and the stack size can be easily guessed), are times when using threaded methods make sense.

8.6 Exercises

Master-slave structure: An appliction requires the main chare to create a large number of chares, and collect a result (lets say an integer) computed by each chare. This structure is sometimes called a master-slave structure.

(a) How suitable is a sync method for this purpose? I.e. can each slave chare's constructor be a sync method that the main chare invokes? Explain your answer.

(b) How will you implement this using futures?

(c) Implement this using explicit suspend/awaken calls, and using a method called "response" in the main chare.

(d) compare the above solutions for their elegance and performance.

Breaking a up a long entry method: An entry method E in your program takes a long time (say hundreds of miliseconds). You want to ensure that higher priority entry methods not belonging to this entry method (e.g. work on the critical path) are executed when they are available. You have made sure no other methods will invoke the chare object that E belongs to. One way of dealing with this issue is to make E a threaded method, and to suspend and awaken it every so-many iterations (say, every 500 usecs).

```
#include "Median.decl.h"

/*readonly*/ int K;

class Main: public CBase_Main {

private:
  CProxy_Partition partition_array;
  double median;

public:
  Main(CkArgMsg* m) {
    if(m->argc < 4){
      CkAbort("\nUsage: ./median [num_chares] [numbers_per_chare] [suggested median]\n");
    }
    int num_chares = atoi(m->argv[1]);
    K = atoi(m->argv[2]);
    median = atoi(m->argv[3]);
    delete m;

    partition_array = CProxy_Partition::ckNew(num_chares);
    thisProxy.computeMedian();
  }

  void computeMedian() {
    int iteration = 0;
    double min_range = 0.0;
    double max_range = 1.0;

    do{
      CkReductionMsg* msg;
      partition_array.queryCounts(median, CkCallbackResumeThread((void*&)msg));

      int *counts=(int *) msg->getData();
      int numSmaller = counts[0];
      int numLarger = counts[1];
      double error = (double)abs(numSmaller-numLarger)/(numSmaller + numLarger);
      if(error < 0.01) break;

      if(numSmaller > numLarger)
           max_range = median;
      else min_range = median;

      median = (min_range+max_range)/2;
      iteration++;
    }  while(true);

    CkPrintf("\nResult calculated median = %lf (in %d iterations)\n", median, iteration);
    CkExit();
  }
};

```

Figure 8.9: Finding median using threads: class Main in the C++ file `median.C`

```
53 class Partition: public CBase_Partition {
54
55   public:
56     double *numbers;
57
58     Partition() {
59       numbers = new double[K];
60       srand48(time(NULL));
61       for(int i=0;i<K;i++){
62         numbers[i] = drand48();
63       }
64     }
65
66     Partition(CkMigrateMessage* m):CBase_Partition(m){
67     }
68
69     void queryCounts(double median, CkCallback &cb){
70       int counts[2];
71       counts[0] = counts[1] = 0;
72       for(int i=0;i<K;i++){
73         if(numbers[i]<median)
74           counts[0]++; // # smaller than median
75         else
76           counts[1]++; // # larger than median
77       }
78
79       contribute(2*sizeof(int), counts, CkReduction::sum_int, cb);
80     }
81 };
82
83 #include "Median.def.h"
```

Figure 8.10: Finding median using threads: class Partition in the C++ file `median.C`

Chapter 9

Commonly Used Constructs

New organization:

(I don't like the "new organization. I am goiung to make chapters here.

One for threads (maybe with simpler/fewer examples)

one with additional constructs: messages, priorities, QD and sections, and dynamic insertion/deletion/on-demand.. (that is a bit disparate, but its ok for now).

and one: acknowledging physical: groups, nodegrops, [initnode/initproc], array maps.

The topic covered so far in this book, chares and their arrays, SDAG, and load balancing, are sufficient to write most applications in Charm++. However, before we devote a chapter on designing parallel applications in Charm++ (Chapter 12), this chapter introduces a few more commonly used constructs in Charm++. We have found these features to be of significant aid in improving programmer's productivity and application's performance. These features act as productivity enhancers because they provide routinely desired functionalities that need not be reimplemented in every application. At the same time, some of them lead to improved performance as efficient interfaces and implementations are provided by the runtime system.

9.1 Callback

Callbacks were briefly introduced in Section 3.3. In Charm++, a callback is an action that should be performed when a condition is met. The action may be execution of simple C++ function, or an invocation of an entry method on a specific chare; it can even be a broadcast to a certain chare array. Given that most operations in Charm++ are asynchronous, a callback can be targeted via several conditions: completion of a reduction, end of a timer, completion of load balancing (for which the entry method ResumeFromSync is the callback), etc.

The class CkCallback is defined by the Charm++ runtime system with a wide range of constructors. The most commonly used among them is used to create a callback that invokes an entry method on a set of chares:

```
CkCallback cb(CkIndex_ClassName::entryMethod(NULL, ...), proxyToChare(s));
CkCallback cb(CkReductionTarget(ClassName, entryMethod), proxyToChare(s));
```

CkIndex_Foo is yet another system defined class for every chare class Foo. For every entry method, EM, in a chare class Foo, CkIndex_Foo has a member function named EM. A call to the method in CkIndex_Foo with NULL as an argument returns an integer that helps Charm++ RTS uniquely identify the member function of Foo across nodes. As a result, when a callback is created as shown above, the Charm++ RTS knows the entry method that should be invoked, and uses the proxy provided in the constructor to identify the targeted chares.

The second version, which leverage CkReductionTarget, should be used for creating a callback to an entry method that is marked as reductiontarget in the ci file, i.e., the method are being used as target of a reduction completion. For historical reasons, Charm++ assumes reductions to always deliver a *message* to their targets (refer Section 9.2). Use of reductiontarget keyword in the ci file removes this restriction, and lets users specify a parameter list of basic data-types for an entry method that is a target of a reduction.

9.2 Messages

So far, we have described communication between chares as invoking methods on a proxy with some parameters. Internally, Charm++ RTS must serialize or marshal these parameters; the parameters are traversed and copied into a message which is sent over the network. On the receiving end, the message is deserialized and the method is invoked by the Charm++ RTS. Although sending parameters is very natural from a programmer's perspective, it incurs the overhead of traversing the parameters and copying them in and out of a message.

When sending large buffers such as an array, the extra copying the system performs can be expensive. If the data is in a flat C++ array and it is declared as such in the .ci file, the system will eliminate one of the copies at the receiving end. This is because the array size is known and the structure is flat, which lets the system pass a pointer to the method from the internal message that is generated. However, the copy on the sending side remains.

To completely eliminate any system copies, the programmer must explicitly create a message and send it. Thus the programmer is responsible for filling the data inside the message on the send side, and pulling the data out on the receiving end. The size of a message must be known at allocation time. Under the hood, Charm++ will allocate the size requested by the user but add bytes for the internal fields that it uses to keep track of the message and deliver it properly.

Use of messages also allow data to be resent after it is possibly modified. So if a chare receives a message, it can modify the message contents, and immediately resend it to another chare without copying.

Messages in Charm++ must be flat structures or arrays that are defined inline (size known at compile time), or should contain variable-size arrays that can be declared in the .ci file for that message. If the message does not have a variable-size array, it can be declared as follows in the .ci file (where <MessageType> is a name the programmer provides):

```
message <MessageType>;
```

There must be a corresponding class in the C++ code:

```
class FooMsg : public CMessage_FooMsg> {
public:
  // fields
  int bar;
  char foobar;
};
```

Note that, while a chare class, Foo, has to inherit from the system defined class CBase_Foo, a message class, FooMsg, has to inherit from the system defined class CMessage_FooMsg. Otherwise, declaring a message is as simple as the above code. Once it is declared, it can be used in entry methods as follows:

```
entry void method1(FooMsg* msg);
```

Note that you always use a pointer to the message when it is sent or received. The corresponding C++ code for receiving this message could be as follows:

```
SomeChare::method1(FooMsg* msg) {
  int receivedBar = msg->bar;
  // ...
  delete msg;
}
```

You are responsible for deallocating a message that a chare receives! The system does not garbage collect the message (except for special cases discussed later), so the programmer must deallocate it, just as the programmer allocates a message as shown below. Since the deallocation can be done anytime, you can keep a pointer to it and store it for a while.

To send a message to a chare, the message must be allocated (possibly filled with data) and then sent as shown below:

```
FooMsg* msg = new FooMsg();
msg->bar = 5;
msg->foobar = 'a';
proxyToSomeChare.method1(msg);
```

You are not responsible for deallocating a message a chare sends!. Once an

entry method has been called using a message, the Charm++ RTS owns the message, and deletes it when it is safe to do so.

9.2.1 Variable-Size Arrays in Messages

Fixed-size messages are easy to define, but have limited use cases. Being able to put a variable-size array in messages increases the utility of messages significantly . This can be done by telling Charm++ the name and type of the array (in the .ci file) and then declaring that same name and type in the C++ code. You also need to pass the size of the array during the allocation, so Charm++ knows how much memory it should allocate for the message.

Array are declared in the .ci file as follows:

```
1  message <MessageType> {
2    int* foo;
3    char* bar;
4  };
```

The corresponding declaration in the C++ code could be as follows:

```
1  class BarMsg : public CMessage_<MessageType> {
2    int* foo;
3    char* bar;
4  };
```

On the sending side, a message has to be allocated using an overloaded C++ new operator that takes parameters that are the size of the variable sized arrays in the order these arrays are defined in the .ci file.

```
1  BarMsg* msg = new(100, 200) BarMsg();
2  memcpy(myLocalFoo, msg->foo, 100 * sizeof(int));
3  memcpy(myLocalBar, msg->bar, 200 * sizeof(char));
4  proxyToSomeChare.method3(msg);
```

On the receiving end, the BarMsg::foo and BarMsg::bar fields can be accessed like any normal C++ array. Note that messages will often contain the size of arrays so that the receiver knows the length.

Example: We now provide an example that gives a concrete illustration of the declaration, definition, allocation and deallocation of variable-size messages in a Charm++ program. For this purpose, we turn to a simple molecular dynamics simulation with force computation added to the one covered in Section **??**, instead of the random perturbation. However, the primary focus is the use of explicit messaging to communicate particles between chares instead of using parameter marshalling.

Recall that we are given a (large) set of identical atoms enclosed within a simulation volume. Here, the atoms are not bonded together, and are assumed to have no electrostatic

```
1  struct ParticleMsg : public CMessage_ParticleMsg {
2    Vector3D<rtype> *positions;
3    Vector3D<rtype> *forces;
4    int nParticles;
5
6    int source;
7    int iteration;
8
9    ParticleMsg(int src, int iter, int np) :
10     source(src), iteration(iter), nParticles(np) {}
11 };
```

Figure 9.1: C++ code - Structure of message used for communication particles.

```
1  message ParticleMsg {
2    Vector3D<rtype> positions[];
3    Vector3D<rtype> forces[];
4  };
```

Figure 9.2: ci code - variable size message declaration.

interactions: we intend only to calculate the van der Waals forces between them, and thus compute the trajectories of individual atoms. We won't discuss the actual decomposition of particles onto chares, but simply note that each chare is assigned a subset of particles that is disjoint from every other chare's subset. Furthermore, the union of these subsets is the input set of particles.

To compute the forces, the chares (called Worker) form a ring. In the beginning, every chare passes the position of its atoms and null forces to the next chare in the ring. The receiving chares compute the forces between these atoms and their atoms, update the forces, and forward the position and the forces to the next chares. In this way, after a fixed set of iterations, one can compute the total forces on all the atoms due to all other atoms.

First, we define a `ParticleMsg` that will hold a subset of particles to be communicated between chares. The structure of this message is shown in Figure 9.1. It contains two (variable-length) arrays: `positions`, which holds the positions of some subset of particles, and `forces`, which holds the accumulated forces on the particles in that subset. Both arrays are of type `Vector3D<rtype>`, where `rtype` denotes the type of each component of the vectors (likely `float` or `double`). The message also has an integer field, `nParticles`, which holds the number of particles in the subset. The `.ci` code that designates `positions` and `forces` as variable-length arrays is shown in Figure 9.2.

Next, we look at how to instantiate a `ParticleMsg` with a requisite amount of space for

```
1 // in Worker::start(), send own particles to neighbor
2 ParticleMsg *msg;
3 msg = new (numParticles,numParticles) ParticleMsg(thisIndex,iteration);
4 for(int i = 0; i < numParticles; i++){
5   msg->positions[i] = locals[i].position;
6   msg->forces[i] = Vector3D<rtype>(0.0);
7 }
8 thisProxy[next].compute(msg);
```

Figure 9.3: Instantiation and initialization of messages.

```
1 // in Worker::compute(),
2 // when done calculating forces on local particles due
3 // to received particles, forward them to ring neighbor...
4 thisProxy[next].compute(msg);
5
6 // ...or, if accumulated forces are to be returned to the
7 // source chare, i.e. the owner of the particles, send
8 // them back:
9 thisProxy[msg->source].forces(msg);
```

Figure 9.4: Reusing a message to forward it to the next chare.

particle position and force coordinates in Figure 9.3. A chare would instantiate a `ParticleMsg` when sending its particles to its neighbor in the communication ring. This happens in method `Worker::start`, which marks the beginning of the work to be done. Note that we only discuss the relevant code pieces here for this example; for the complete example code, please refer to the code provided with this book.

Line 3 of Figure 9.3 shows the allocation of a `ParticleMsg` with `numParticles` position vectors and `numParticles` force vectors. Thereafter, the positions of the particles owned by the chare are copied into the message, and the force values therein are initialized. Note that `struct ParticleMsg` has data members `source` and `iteration` to replace the marshalled parameters that are received by `Worker::compute` as input. Finally, the chare sends this message to the `compute` method to its neighbor in the communication ring (line 8). Since `Worker::compute` now expects to receive a `ParticleMsg`, its signature is as follows: `void Worker::compute(ParticleMsg *msg);`

Of course, particles are communicated in other contexts as well: when a chare receives a set of particles from one of its ring neighbors, it must "consume" them to calculate forces on its local particles due to these received particles, and having done so, must pass them on to its other neighbor. Moreover, once a subset of particles has been "consumed" in this manner

by half the chares in the ring, it must be sent back to its owner, which will have accumulated forces on its local particles due to the *other* half of the ring. This is simply doing by reusing the message as shown in line 4 and line 9 of Figure 9.4. Notice how we are able to *reuse* the message passed into the method `Worker::compute`, unlike in the parameter-marshalled version of the code. In the latter, the marshalling of the user-provided `array` incurs copying overhead, making the message-based version more efficient.

9.2.2 When should you choose messages over marshalled parameters?

The performance-conscious programmer will choose messages over marshalled parameters in the following situations. Marshalled parameters are available to your code only in the scope of the entry method in which they are received. Therefore, if a chare needs to access marshalled parameters after the entry method has finished, they must be copied into heap-allocated buffers. This often happens in programs in which a *chare receiving data may not be ready to process the data immediately upon receipt*. In this case, data received via a message can be saved for later use by simply recording the pointer to the input message received by the entry method. This is often more efficient than copying received marshalled parameters onto the heap, especially if the entry method receives arrays and large, composite data structures. Of course, you must remember to `delete` received messages yourself: Charm++ will not do this for you, except in special cases.

Messages are also helpful in avoiding an extra sender-side copy when the data to be sent in created explicitly for communication or serialized explicitly by the user. For example, in the molecular dynamics code, if the chare has to only send atoms with a particular property, the user code will have to traverse the set of atoms residing on this chare, and create a new vector to be sent as a parameter, which is then copied into a message by the runtime. Instead, this new vector could be part of a message and thus avoid an extra copy at the sender-side. Note that you must allocate space for the vector inside the message when the message is allocated, which means you must know the (at least an upperbound on the) size needed at allocation.

You may also benefit from using a message instead of marshalled parameters if your entry method exhibits the following pattern: it receives data of a certain size, alters the values in the body of the entry method, and sends it to another chare, such that the size of sent data is the same as that of the received data. In such a case, the contents of an incoming message could be altered, and the message *reused* in the invocation of an entry method on some other chare. For instance, the ring communication example from Section 3.1 would benefit from using messages: instead of accepting a number of marshalled `int` type parameters, the `Ring::doSomething` method could accept a message, alter its contents to reflect the number of chare array elements yet to receive the message, and send *the same* message to the next chare in the ring.

Note that when user code passes a message to the Charm++ runtime system, e.g. through

an entry method invocation, the memory associated with the is no longer available to the user. In particular, attempts to dereference a pointer to a message object that has already been used in an entry method invocation, will yield incorrect results and possibly cause your program to crash. Therefore, a received message can only be reused *once* at the receiving end, i.e. it can only be passed as input to a *single* entry method invocation by the chare that received it, although that invocation may be a point-to-point communication, or a broadcast/multicast. However, a received message *cannot* be used in multiple entry method invocations at the receiving end. For instance, if you were writing a Charm++ program in which the contents of a message were being passed down the nodes of a spanning tree constructed from an array of chares, the code that forwards a message received by an internal node could send the received message to only *one* of the chare's spanning tree children; for all other children, new messages would have to be allocated.

9.3 Priorities

9.4 Quiescence Detection

Consider a naive parallel program for computing shortest distance of every vertex from a root vertex in a graph G. Let us assume that the graph G is large, and vertices of G are divided among various chares. A simple implementation would start from the chare R that contains the root vertex, and send distance 1 messages to all its immediate neighbors. Subsequently, whenever a chare receives a distance update message for one of its vertex, it checks if the new distance, d, is less than the distance currently assigned to the vertex. If the new distance is lower, distance $d + 1$ messages are sent to immediate neighbors of the vertex; otherwise, the chare does not do anything.

For this naive but common implementation described in the previous paragraph, a critical question arises: when does the computation end? The answer is when none of vertices send out distance messages, i.e. there is no outstanding message in the system, either in transit or waiting to be executed. This quiet state is termed Quiescence in Charm++. We have found that many applications have scenarios in which detecting quiescence efficiently is important. For example, in adpative mesh refinement, quiescence is needed to ensure that restructuring of mesh and exchange of particle is finished. Similarly, in parallel sorting, quiescence ensures that redistribution of the data is complete.

In Charm++, quiescence detection can be triggered by the following call:

```
CkStartQD(callback);
```

In the CkStartQD call, callback is the action that should triggered when quiescence has been detected. Every CkStartQD call is registered independently, i.e. to specify one action, CkStartQD should only be called from one chare. As soon as CkStartQD is called, the control

```
1  mainmodule taskSpawn{
2    readonly CProxy_Driver driverProxy;
3    readonly int delay;
4
5    mainchare Driver {
6      entry Driver(CkArgMsg *m);
7      entry void results();
8    };
9
10   chare Worker {
11     entry Worker();
12   }
13 };
```

Figure 9.5: ci code for quiescence example, in which tasks are spawned from the mainchare.

returns to the calling function and the Charm++ RTS begins the process of detecting next quiescence. The process of detecting quiescence overlaps with the rest of the program execution. Eventually, when the RTS detects quiescence, it invokes the callback registered with the CkStartQD call.

In Figures 9.5 and 9.6, we present an example program to demonstrate use of quiescence detection in a realistic scenario. In this example, we have a mainchare, Driver, and a chare class called Worker (see Figure 9.5). The execution of mainchare's constructor creates many Worker chares which are expected to perform some work. This represents a generic master-slave scenario, in which the master assigns work to a set of workers. How does the master know when all the workers have completed their work? As shown in Figure 9.6, this can be easily achieved by triggering a quiescence detection from the mainchare with a callback to one of its entry methods.

9.5 Dynamic insertion, deletion, on demand.

9.6 Sections and Delegation

It happens often that a subset of chares in a chare array need to be addressed as a unit. For example, when vertices of a graph are partitioned among chares, a chare may need to send common information to all other chares that are its vertex-neighbors. In such scenarios, it is useful to be able to create groups or subsets. In MPI, creation of sub-communicators provides such a functionality. Chare array sections, or sections in short, provide similar capability in Charm++.

Similar to chare array proxies, sections are addressed via section proxies. For every chare

```
1  #include "taskSpawn.decl.h"
2
3  CProxy_Driver driverProxy;
4  int delay;
5
6  class Driver: public CBase_Driver {
7    int count;
8    int tasks;
9    double startTime, endTime;
10
11 public:
12   Driver(CkArgMsg*m){
13     startTime = CkWallTimer();
14     driverProxy = thishandle;
15     tasks = atoi(m->argv[1]);
16     delay = atoi(m->argv[2]);
17
18     count = tasks;
19     for (uint64_t i=0; i<tasks; ++i)
20       CProxy_Worker::ckNew();
21
22     CkCallback endCb(CkIndex_Driver::results(), thisProxy);
23     CkStartQD(endCb);
24   }
25
26   void results() {
27     endTime = CkWallTimer();
28     CkPrintf("Total execution time: %.2f s\n", endTime - startTime);
29     CkExit();
30   }
31 };
32
33 class Worker: public CBase_Worker {
34   public:
35     Worker(){
36       double volatile d = 0.;
37       for (uint64_t i=0; i<delay; ++i)
38         d += 1. / (2. * i + 1.);
39     }
40 };
41 #include "taskSpawn.def.h"
```

Figure 9.6: C++ code to demonstrate use of quiescence detection.

array, Foo, the system defines a section proxy class called CProxySection_Foo. A new section can be created and a proxy to it can be obtained by calling ckNew on the system defined class CProxySection_Foo. In the most common form, ckNew call for section creation accepts three arguments: the array ID of the chare array, the indices of the array elements that should be in the section, and the number of the elements. The array ID of a chare array is obtained by called member function ckGetArrayID on the chare proxy. Here is an example of section creation:

```
1  //array to store indices of array elements of a 3D chare array
2  CkVec<CkArrayIndex3D> elems;
3  for (int i = 0; i < 10; i++)
4    for (int k = 0; k < 10; k++)
5      elems.push_back(CkArrayIndex3D(i, 0, k));
6
7  CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, elems.getVec(), elems.size());
8  //generalized form
9  CProxySection_Foo proxy = CProxySection_Foo::ckNew(fooID, <vector of array elements>, <number
       of array elements>);
```

This example shows us how to create a section of array elements whose y-index is zero in a 3D chare array of dimensions $10 \times 10 \times 10$. The other useful variant of ckNew that can be used to create sections lets user specify the range of indices that should be part of the newly created section. The following code creates the same section as above but by using the alternate form of the ckNew call:

```
1  CProxySection_Hello proxy = CProxySection_Hello::ckNew(helloArrayID, 0, 9, 1, 0, 0, 0, 9, 1);
2  //generalized form
3  CProxySection_Foo proxy = CProxySection_Foo::ckNew(fooID, [ (lower, upper, step), ...]);
```

9.6.1 Delegation

While creation of sections enables addressing a subset of chares as a unit for operations such as broadcast, the Charm++ RTS provides a delegation framework to customize implementation of various operations on them. In particular, we will look at CkMulticast library that optimizes multicast and reduction operation on sections. To use this library, a user should create a chare array of type CkMulticastMgr that is defined by the Charm++ RTS. This chare array has one chare per processing element (PE); see Groups in Section 10.3. Once created, the user can explicitly delegate the task of performing broadcast and reduction on a section to this chare array. Here is a concrete example of how the delegation should be done:

```
1  //assume we have created a section proxy called fooSectProxy
2  CkGroupID mCastGrpId = CProxy_CkMulticastMgr::ckNew();
3  //obtain a local handle
```

```
4 CkMulticastMgr *mCastGrp = CProxy_CkMulticastMgr(mCastGrpId).ckLocalBranch();
5 // delegation - to be done only once
6 fooSectProxy.ckSectionDelegate(mCastGrp);
```

As with chare array creation discussed in previous chapters, line 2 shows the creation of a array of the system defined class CkMulticastMgr. Here, the value returned by ckNew is stored in a variable of type CkGroupID, which is the base class from which all proxy classes inherit. In line 4, the function call ckLocalBranch() is used to obtain a pointer to the local object of the chare array, which is needed to perform the delegation. Finally, ckSectionDelegate is used to tell the runtime system that any call to the given section proxy should be delegated to the aforementioned chare array of type CkMulticastMgr. As a result, when a future invocation is performed on the section proxy, e.g. fooSectProxy.recvData(), the runtime system informs the multicast manager of the multicast request. Under the hood, the multicast manager performs this multicast by taking topology and other performance related aspects into account. At the very least, it performs the multicast by using a tree-based algorithm, and sends only one message to every PE on which the members of the section reside.

We advice the readers to visit the Charm++ manual here to learn the usage of CkMulticastMgr for performing efficient reductions.

Chapter 10

Acknowledging the Physical: Groups, Nodegroups, Maps, etc.

In the Charm++ constructs seen so far, any aspects of the physical parallel machine were almost entirely absent. In fact, it is a strength of the Charm++ model that the programmer can afford to ignore the physical, and focus solely on the logical structure of their application. Explicit grainsize considerations were one of the few intrusions of the "physical" (i.e. the overhead per message) that we allowed. In addition, we had a call to find out which processor the call is coming from, `CkMyPe`, which we used only for diagnostic or illustration purposes.

However, there are situations where optimizing the program requires some knowledge of the physical architecture of the machine, and the ability to program to it. We provide the constructs needed for that in this chapter. We do this with some trepidation, because excessive and unnecessary use of these constructs will come in the way of adaptivity provided by the runtime system. So, use these constructs with caution, and only when you are convinced that other abstractions, and especially libraries, provided by the Charm++ system are inadequate for your purpose.

10.1 Charm++ view of the physical machine

In the view supported by Charm++, the machine consists of a set of logical nodes, and each node consists of a set of processing elements (PEs). Each chare is anchored to a particular PE. All the chares belonging to a logical node (aka OS process) have access to a common shared memory. In practice, a logical node is an operating-system level process. A physical host (physical node) is largest unit which such a logical node might incude. In other words, a single host may be split into multiple logical nodes (e.g. a NUMA domain), each one of which runs a single OS process, housing multiple PEs. A PE is associated with a single scheduler. The physical node may include many *cores* each with multiple hyperthreads (or

symmetric multi-threads). A subset of these constitute the set of PEs. The choice of how many processes to run on a physical host, and how many PEs to create within each process is made by default by the system, with override provided to the end-user via command line options. [1]

Thus, there are three distinct views: First, there is a pure Charm++ view presented by the previous chapters. Here, the program consists of a multiple indexed collections of chares that communicate with each other via asynchronous method invications, and participate in collective communication operations such as reductions and broadcasts. No awareness of the physical machine is needed here. The second is the view of the physical architecture provided from inside a (impure) Charm++ program: This includes (logical) nodes (aka processes), and PEs. A chare is aware of which PE it belongs to and which node it belongs to, and has constructs to exploit this knowledge. We will see what these constructs are in the sections below. The third view is the real physical view: a set of hosts, each of which includes some number of cores, each of which supports (potentially) multiple hyperthreads.

Figure **??** shows the last 2 views overlapped on each other.

The reason that we don't always have a one-to-one mapping of processes to hosts is that it is sometimes more efficient to divide them further. This may be because the host consists of multiple sockets (processor chips), and memory accesses are non-uniform for the 2 sockets: i.e. cores belonging to one socket may take longer to access memory belonging to another socket. Further, certain types of communication related tasks in the runtime system may become bottlenecks if they are catering to a very large number of PEs.

The reason that we don't always have a one-to-one mapping of PEs to hardware threads (or cores) is that the choice depends (prmarily) on memory bandwidth issues. For an application that accesses a large amounf of memory in a fashion that requires constant movement of data between memory external to the processor chips (e.g. DRAM) and the procesor chip, we should to use fewer PEs, say one per core. This is because the increased parallelism by using more PEs is not going improve performance, which is limited by the total bandwidth between processor chip and memory. In contrast, an application with a lot of data reuse (or with small memory footprint) can use more PEs, say putting a PE on each hardware thread.

Normally, you should let the runtime system make these choices. It makes pretty good choices by default, and its ability to change those choices at runtime is being improved in successive versions of the runtime system. When you are aware of the need to optimize performance by taking these decisions yourself, you can do it by simply experimenting with a few alternative configurations. Section **??** describes how to specify such configurations.

[1]* The relevant command line options for explicit control include *++ppn, +pemap*, and `+commap` options, as well as semi-automated variants such as ++oneWthPerCore.

10.2 Array maps

As you know, the chares that are members of a chare array are assigned to processors by the runtime system. But how does that work? Initially, the system uses a default mapping method to do the assignment, and gets the chares started on those processors. Afterwards, the dynamic load baalncer may migrate them across processors. The default map for a one dimensional array may be a block map, where each processor gets a contiguous sets of chares, for example. I.e. Let us say there are 2000 chares and 10 hosts with 10 cores each, and it has been decided (either by the runtime system or by the user via command line arguments) there are to be 2 processes per host, and 5 PEs on each process. PEs are numbered contiguously within a node. So, PE 0 (on process 0) would have chare array elements 0..19, PE 1 (also on process 0) would have chare array elements with indices 20..39, etc.

For higher-dimensal arrays, the default system maps also use some for of blocking, as above. The processor that a chare is initially mapped to is called its *home* processor. The RTS uses the knowledge for the home processors for a variety of purposes, including for tracking the location of a chare as its migrates.

Now, for application-specific reasons, you may (hopefully rarely) want to control this initial mapping yourself. For example, you have a 3-dimensional array of chares, and you know that the chares communicate a lot more with chares along the 3rd dimensions (say Z dimension) than they communicate along the other two. You don't have dynamic variability in application, so you don't expect (at least this array) to be influenced by dynamic load balancing. So, you can want to override the system's default mapping.

Charm++ provides a way of associating map function with each chare array. This function takes the indices of a chare array element and returns a PE number. Although the initial mapping is done, well, only initially, the procNum function is called many times during the execution of the program (to find the home processor for location information, for example); And since the procNum function may depend on stored state in some rare cases, you acually are required to specify a mapping *class*, rather than just the procNum function. So, in this exmaple, you will write a class (say, `MyMap`) that is a subclass of `CkArrayMap`, and write its procNum function . `CkArrayMap` is a *group*: that means one element of it exists on each processor, so the runtime system can call this function on any processor when it needs to find the home processor. Then, before creating the chare array that is to be mapped using this function, you create a group of type MyMap and set it as the mapping group in CkOptions (remember CkOptions from Chapter **??**?).

For this example (See Figure **??**), we define a procNum function that linearizes the 3 dimensions and then maps the linear index in block fashion. This (given the way we are linearizing the indices) ensures that chares with the same X and Y (1st and second) index will tend to be on the same processors (and, since PEs of a node have contiguous processor numbers, the chares on the same node will tend to have the same X,Y index).

```
class MyMap: public CkArrayMap....

register(...)
int procNum(hdl..idx.....)
{..
int linearizedIndex = M*N*index.x + N*index.y + index.z;
int pe = linearizedIndex/CkNumPes();
return pe;
}
```

Say something about this.

And before creating a chare array, of type (say) `A1`.

```
 //Create the map group
  CProxy_BlockMap m=CProxy_MyMap::ckNew();
//Make a new array using that map
  CkArrayOptions opts(nElements);
  opts.setMap(myMap);
// Set other options, such as callbacks etc.
  a1=CProxy_A1::ckNew(parameters,opts);
```

Notice that the function uses (and needs to have access to) information about the dimensionality of the array, the knowledge that it is a dense, rather than sparse, chare array, and the knowledge of the total number of PEs in the system.. This last piece is the minimum knowledge of the "physical" aspect, which is obtained using the function CkNumPEs. More sophisticated procNum functions may want to access more detailed information about physical topology, including the interconnection network topology, and number of PEs per node, etc. These function are found in the Charm++ manual (see, for example, the section titled *Querying Network Topology*).

10.3 Groups

We already saw how we create a Map as a group, in a simple example. Let us explore this contruct in more detail.

A group is like a chare array with 2 differences: (a) There is exactly one member of it on each PE; Therefore, of course, the member chares are not migratable. (b) Unlike chare array elements, it is possible to get a regular C++ pointer to the *local member* (sometimes called the local *branch* of a group) via a special function: `CkLocalBranch(groupProxy)`, where

groupProxy is A proxy to the group. A proxy for the group is created just as it is created for a chare array: it is a return value for the CkNew call, and it can be passed around as a first class object.

The ability to get a local pointer is the special glue that makes a group a very versatlie function, especially for writing "services" or for creating interfaces between modules. Multiple chares, belonging to multiple different collections of chares, can connect to the single representative of the group on a given PE. The interaction pattern is shown in figure [**?**].

we need to draw a figure here.

10.3.1 Distributed Tables and Processor Level Caches

As an example of the use of group, let us consider the following situation: we have an iterative simulation program expressed with a single chare array. From time to time, each chare array elelement needs some information, based on the current features of the local chunk it is simulating. (May be these are coefficients of how the material behaves under specific stress situations in a strucral dynamics simulation.. .) Each property-set is associated with a *key*. The set of such properties is too large to fit their table on a single processor or even a single node. So, it has be distributed across processors. The abstraction is called a *key-value store.*

This store can be implemented as a chare array. When you want the value for a particular key, you hash the value to an index and send the request to that element of the keyValue chare array. (It will eventually send a response with the value. To be able to tell that response from other responses for other keys you have sent, it can (say) send the key also along with the response.

Figure 10.2 shows the portion of the program that implements this idea. .

The kvstore is a one-dimensional chare array. In its constructor, each element computes the values for each of the keys it is responsible for, and stores them in an internal table. In a general solution, such values may be read from disk, or computed on demand. Also, the set of keys associated with each chare array element may have to be determined by a more complex process. Here, for the sake of a simple example, we assume the keys constitute a contiguous set of values that are partitioned to different chare array elements via a simple block partitioning scheme. Thus, if there are hundred thousand keys, the will range from 0..99,999, and if there are hundred chares, chare 0 on keys numbered 0 to 999, etc.

The client is a one-dimensional chare array. In its seek entry method, it simply randomly decides which keys it needs values for, and sends lookup requests to the kvStore chare array element responsible for each of those keys. EXPLAIN 3 parameters. The values are sent back to it via the process (??? Change the name) method.

We now notice that it is possible that multiple chares on the same PE may request the property values for the same key. We don't want to create network traffic for such duplicate requests. How can we do that? Notice that we are talking about a processor-level concept here: avoiding duplicate requests from the same *processor*. (The chares themselves are not

```
1  mainmodule kv {
2    readonly int numbuckets;
3    readonly CProxy_kvstore stores;
4    readonly CProxy_client clients;
5    readonly CProxy_middle mid;
6    mainchare main {
7      entry main(CkArgMsg*);
8    };
9    class CProxy_kvstore;
10   class CProxy_client;
11   group middle {
12     entry middle();
13     entry void lookup(int,int);
14     entry void process(int,int,int);
15   };
16   array [1D] client {
17     entry client(int,int,CProxy_main);
18     entry void seek();
19     entry void process(int,int);
20   };
21   array [1D] kvstore {
22     entry kvstore(int,int);
23     entry void lookup(int,int,int);
24   };
25 };
```

Figure 10.1: Key-Value Store: interface file kv.ci

```
31 class kvstore : public CBase_kvstore {
32   std::vector<int> data;
33 public:
34   kvstore(int maxkey,int maxval) {
35     std::mt19937 engine(thisIndex);
36     std::uniform_int_distribution<> distro(0,maxval);
37     for (int i=thisIndex;i<maxkey;i+=numbuckets) {
38       data.push_back(distro(engine));
39     }
40   }
41   void lookup(int key,int senderidx,int clientidx) {
42     int idx=key/numbuckets;
43     mid[senderidx].process(key,data[idx],clientidx);
44   }
45   static int hash(int key) {
46     return key%numbuckets;
47   }
48 };
49 class middle : public CBase_middle {
50   std::unordered_map<int,int> cache;
51 public:
52   middle() {}
53   void lookup(int key,int senderidx) {
54     auto iter=cache.find(key);
55     if (iter!=cache.end()) {
56       clients[senderidx].process(key,iter->second);
57     }
58     else {
59       stores[kvstore::hash(key)].lookup(key,thisIndex,senderidx);
60     }
61   }
62   void process(int key,int value,int clientidx) {
63     cache[key]=value;
64     //send data back to source client
65     clients[clientidx].process(key,cache[key]);
66   }
67 };
```

Figure 10.2: Key-Value Store: implementation file kv.C

aware of the other chares on the same processor.). The solution is to use a `group` that will cache the responses and mediate the requests.

We will not show the entire example code, but just the group definition. The group has a locally accessible (public) method called `lookup`, while the client has a method called `process` that it expects to be invoked with the value associated with the given key. If the requested key is locally available, the group invokes the method, while if it is not, it forwards the request to the key-value store.

10.4 Node-Groups and Effectively Utilizing Shared Memory

Chapter 11

SHM: This definitely needs a chapter of its own

11.1 Execution in Shared Memory Mode

Charm++ is designed to work efficiently and portably on a signle (as in your desktop) shared-memory nodes, clusters of single-core nodes as well as clusters of multi-processor (SMP) nodes. The base model in Charm++ requires you to encapsulate the data in Chares. This is good for locality. The techniques and primitives described in this section exploit shared memory within a node without sacrificing this locality-promiting property of Charm++.

Given the increase in the number of available cores on a physical node, parts of the Charm++ runtime system have been redesigned to take advantage of modern multicore systems. On these systems, Charm++ should be run in a shared memory mode (SMP mode), wherein the RTS creates a set of *worker* threads and a *communication* thread. The worker threads are typically tied to hardware resources such as cores, and are used to execute entry methods on chares, etc. The communication thread, in contrast, is used to progress the underlying communication engine and send-recv application and RTS messages.

Most of the changes described above do not affect the way Charm++ programs are written and its features are used. However, to avail the benefits of the SMP mode listed in the next paragraph, the end user has to take a few minor steps. First, when building Charm++, the user has to pass smp as an optional build argument. Secondly, when executing the program, an additional argument +ppn <number of cores used by threads of a process> should be passed. This argument tells Charm++ the number threads that should be launched as part of a process executing on a node. Finally, another runtime argument that defines the affinity behavior of these threads should be provided. We refer the reader to the Charm++ manual entry here for more details on this aspect.

The primary advantages of using SMP mode in Charm++ are as follows:

- Messaging within a process, which often spans the entire physical node, is performed by passing data pointers, and thus is faster.
- Communication progress does not interfere with the execution of user program since a dedicated communication thread is used, and vice versa.
- Operations such as broadcast, queries etc are optimized since they are aggregated across all PEs running on the node.
- Within node parallelization constructs, such as OpenMP and CkLoop, can be used to further parallelize computation intensive operations.

11.2 Readonly Variables

Read-only variables in Charm++ provides a special mechanism for sharing data amongst all objects. Their values are initialized once in Charm++'s `main::main()` function, and do not change after that. Each SMP node maintains a single copy of read-only variables, and they can be accessed from any chare on all processors of that node as "global" variables.

Readonly Variables not only supports simple basic data types, but also support complex data types such as arrays and messages ...

We have already used them in examples before.. But here are some more examples, where the data is large, and you have to make a concious design choice to use a readonly structure for it.. (Also, how to make complex pointer-based structures in readonly memory, readonly messages, and use of pack and unpack for complex data structures)

Explain how you can at times break the "readonly" ness if you know what you are doing.

11.3 Conditional Packing

When messages are sent across nodes, messages that contain complex data structure such as pointers need to be packed to a sequential buffer to send over the network, and unpacked on the destination node. While messages sent to the same node in same address space does not require packing and unpacking. This is called *Conditional Packing*. Compiling Charm++ with SMP/Multicore support to take advantage of this feature in which case Charm++ is running multithreaded — one pthread per core .

Conditional packing can be useful for a lot of optimizations to efficiently use shared memory and avoid unnecessay copying overhead. For example, consider the following scenario in a master-slave style application. A master chare assigns a portion of its array exclusively to a worker chare to work on. The worker chare can access and modify its assigned portion of array directly if it is on the same node, but will get a copy of the data if it is on a different node. When the worker chare finishes the computation, it needs to send the result in the

same array back to the master chare via a message. However, when the worker and master chares are on a same node, there is no need to create a message for that purpose since worker thread already directly works on the portion of array assigned by the master chare! Using the conditional packing, we can avoid the extra return message and corresponding copying of the array data if the worker and the master are on the same node, and hence efficiently uses the shared memory. A message is needed only when a result is received from a remote node, where the unpack function is invoked to copy the data from the message right into the array.

The following code illustrate this idea. In this example, master chare — the mainchare creates a number of worker chares that do simple computation on a global array that is stored on the mainchare processor. When creating each worker cahre, the mainchare assigns a portion of the array to it.

```
1  mainmodule pgm {
2
3  readonly CProxy_main mainProxy;
4
5  message packtest_Msg;
6
7  mainchare main
8  {
9    entry main();
10   entry void results(packtest_Msg *m);
11 };
12
13 chare integrate
14 {
15   entry integrate(packtest_Msg *m);
16 };
17
18 };
```

Figure 11.1: Simple master-slave application: the interface file `pgm.ci`

The message-based technique

Marshalling and conditional packing (Filippo's implementation)

Conventions to esnure safety.. (readonly sub-ranges, exclusive-write rights, quicksort example)

We should add the quicksort as an example here.

```
1  #include "pgm.decl.h"
2
3  class packtest_Msg : public CMessage_packtest_Msg
4  {
5    public:
6      int srcpe;
7      int count;
8      double *srcptr;
9      double *ptr;
10     static void *pack(packtest_Msg *);
11     static packtest_Msg  *unpack(void *);
12     packtest_Msg(void) { srcptr=ptr=0; }
13     ~packtest_Msg(void) {}
14 };
15
16 class main : public Chare
17 {
18 private:
19   int count; //Number of partial results that have not arrived yet
20   double startT;
21 public:
22   main(CkMigrateMessage *m) {}
23   main(CkArgMsg *m);
24   void results(packtest_Msg *);
25 };
26
27 class integrate : public Chare
28 {
29 private:
30
31 public:
32   integrate(CkMigrateMessage *m) {}
33   integrate(packtest_Msg *);
34 };
35
```

Figure 11.2: Simple master-slave application:: the header file `pgm.h`

```
1  #include "pgm.h"
2
3  #define ARRSIZE  10000000
4
5  double arr[ARRSIZE];
6  CProxy_main mainProxy;
7
8  void *packtest_Msg::pack(packtest_Msg* m)
9  {
10   packtest_Msg *t = (packtest_Msg *) CkAllocBuffer(m, sizeof(packtest_Msg)+(m->count)*sizeof(doub
11
12   t->srcpe = m->srcpe;
13   t->count = m->count;
14   t->srcptr = m->srcptr;
15   t->ptr = (double*)((char*)t+sizeof(packtest_Msg));
16   memcpy(t->ptr, m->ptr, sizeof(double)*m->count);
17   delete m;
18   return t;
19 }
20
21 packtest_Msg * packtest_Msg::unpack(void *buf)
22 {
23    packtest_Msg *in = (packtest_Msg *) buf;
24    in->ptr = (double*)((char*)in+sizeof(packtest_Msg));
25    if (in->srcpe == CmiMyPe()) {
26      memcpy(in->srcptr, in->ptr, sizeof(double)*in->count);
27      in->ptr = in->srcptr;
28    }
29    return in;
30 }
31
32 main::main(CkArgMsg * m)
33 {
34   int i;
35   int numChares = 10;
36   int size = ARRSIZE/numChares;
37   CmiAssert(ARRSIZE%numChares == 0);
38
39   for (i=0; i<ARRSIZE; i++) arr[i] = i;
40
41   int start = 0;
42   for (i=0; i<numChares; i++) {
43     packtest_Msg *m = new packtest_Msg;
44     m->count = size;
45     m->srcpe = CmiMyPe();
46     m->ptr = m->srcptr = arr+start;
47     CProxy_integrate::ckNew(m, i%CmiNumPes());  // create in round-robin fashion
48     start += size;
49   }
50
51   count = numChares;
52   mainProxy = thishandle; // readonly initialization
53   startT = CmiWallTimer();
54 }
55
```

11.4 NodeGroups

11.5 CkLoop

Chapter 12

Designing Parallel Programs with Charm++

In the chapters so far, we have introduced the essential *concepts and machinery* of Charm++ viz. overdecomposition, chare arrays, SDAG, etc. In most places, we used simple examples meant only to illustrate each concept rather than accomplish a computational task efficiently in parallel. In this chapter, we are going to learn how to wield this machinery for developing parallel applications. Our "applications" will still not be full-fledged parallel application, but they will complex enough to illustrate parallel program design issues.

12.1 Designing a Charm++ Application

When designing a Charm++ application, the first important task is to determine which C++ objects should be chares (parallel objects) and which objects should be plain C++ objects. As described earlier, chares can be thought of abstractly as existings in a "global address space" (Figure 1.2); they exist somewhere in the parallel system, but are not tied to a specific processor. Of course, the runtime will map each chare to a processor, but the program should be written oblivious to actual location of the object. However, for each chare that is created there will be overheads associated. The following are some overheads that should be considered when making parallelizing decisions:

1. Invoking a method on the chare may cause a message to be sent over the network, which requires serializing and de-serializing the message. Serialization is expensive because most of the time all the data must be copied on both ends and may have to be restructured.

2. Every entry method (remotely invokable method) is schedule through the Charm++

runtime scheduler. This means that it will not execute immediately, but instead competes with other entry methods for processor time.

3. The runtime must manage the object's location. The Charm++ RTS will keep a distributed database of the location and if migration is necessary will serialize and deserialize the requisite data in the object. Also, various information may be collected if load balancing is enabled.

Given this, two questions naturally arise: 1) how large should chares be (we call this the grainsize problem)? and (2) which C++ objects should be promoted to chares?

The Grainsize Problem

When we talk about decomposing a computation's work units and data units into objects, how large do we want the objects to be? The short answer is that they are meant to be medium-grained entities. Small enough that the system can do load balancing by moving some of them (for example) and large enough to amortize the overhead of delivering messages (method invocations) to individual objects. Clearly, making each *double* as a chare is an overkill. So, how do you think about the grainsize?

As a working definition, let us define grainsize as the amount of computation per entry-method invocation. Each entry method invocation has associated with it the overhead for queuing it in the schedulers queue, and potential having to serialize and send it to a remote processor. Since each chare has multiple entry methods (and multiple invocations of each entry method), and there may be many chare types in a program, it is useful to define the notion of *average* grain size, and larges grain size.

So, a simple rule of thumb is to define your chare objects in such a way that the average grainsize is sufficiently large compared with the CPU overheads associated with an EP. On most of the computers today, the overheads of the order of a few microseconds. So, if the average grainsize is more than a few tens of microseconds, we are sure the overhead is smaller than a few percent.

On the other hand, if a single grain (e.g. a chare) is too large, it will become a bottleneck. The processor that owns it may become the

INCLUDE THE SCHEMTIC PLOT

(do algebraic equation for execution time as a function of grainsize and overhead, as in the slides?? I think its ok to do it here. So they don't start reading Charm++ with the wrong idea of grainsize like many students do).

Which Objects Should be Chares?

Much of this problem boils down to whether making an object a chare exposes parallel work that is not dominated by overhead. In other words, is the parallel work created greater

than the amount of overhead ensued by sending and receiving messages by this object? This question is complex because by mapping the application well, objects that communicate frequently or with large messages should be placed near each other in the system. So while further parallel decomposition may incur significant communication overheads, the extra parallelism may be well worth it.

The parallelism decision is as much a performance questions as a software engineering one. Often, many different designs will expose different engineering tradeoffs that can be studied in object-oriented literature (beyond the scope of this tutorial).

12.1.1 Design Examples

To get an idea of what we mean by object-based overdecomposition, and to see how Charm++ programs are designed, we next present several examples. These are based on realistic full-fledged applications, but simplified sufficiently that they are easy to understand. The essential ideas we want to impart here are about granularity of parallelism, and about how to think about a parallelizing an application when using Charm++.

Stencil Computation

In a stencil application, we are simulating a n-dimensional grid of values that are updated in discrete timesteps. Typically, each point in the grid will represent a specific value: heat, temperature, pressure, etc. Each point is updated for every timestep based on the nearby values in the grid.

Stencils are typically decomposed into a equal-size chuck for each processor. However, this may not be optimal if the system exhibits any dynamic behavior: frequency variations, faults, etc. In Charm++ we decompose the stencil into chunks that are large enough to fully utilize memory bandwidth and cache, but small enough to be overlapped with other chunks if possible. Using this decomposition, we are able to demonstrate execellent scalability.

NAMD: Biomolecular Simulation

NAMD is a production-level biomolecular simulation package written in Charm++ that has been scaled to many of the largest supercomputers in the world. In 2002, it won the Gordon Bell award for scalability and performance. Since then it has been used for breakthrough science, notably for determining the structure of HIV capsid, which was published in Nature.

The core of the computation in NAMD is the force calculations for the interactions of thousands to millions of atoms. At each timestep, the short and long range forces are calculated, followed by the velocities, and then the positions of the atoms are updated based on this.

In NAMD, a hybrid decomposition is used to increase the amount of parallelism. First, the atoms are decomposed into regional boxes that separate the atoms in the spatial di-

mension. The second partitioning, which is one of NAMD's flagship characteristics, is the force decomposition. Each box interacts with the atoms in nearby boxes. Instead of having each box interact with all the other nearby boxes, an 'Compute' object is created for every nearly box to compute that pair of interactions. The force decomposition unleashes further parallelism that can be effectively exploited, increasing the scalability of the application.

12.2 General Principles of Designing Charm++ Application

A narrative on how to design parallel programs? Identify data, major time consuming portions, decide data decomposition into arrays. Examine for basic pitfalls. (granularity, communication overhead, serialization, bottlenecks)

12.3 Lennard-Jones Molecular Dynamics

Our first design example is based on simulation of atoms (or molecules) under the influence of Van Der Waal's forces. The background for this example is similar to that of the simple Particle Motion example used to demonstrate load balancing in Section **??**.

We have a collection of N particles (atoms), which are not bonded to each other, nor do they carry electrical charge. These particles are spread in a three-dimensional world. In addition to mass, M, each particles has the following dynamically varying attributes: a position (x, y, z) in the 3D space, velocity (v_x, v_y, v_z), acceleration (a_x, a_y, a_z), kinetic energy K, and potential energy P.

The task is to simulate the movement and compute the attributes of particles subject to van der Waals forces. We assume that the particles are inside a 3D box, where we wrap particles from one end to the opposite one. The force that a particle imposes on another particle can be expressed by:

$$F = A/r^6 - B/r^{12}$$

where r is the distance vector between the two particles and A and B are constants. The total force a particle is the summation of the forces from all the other particles within a given cutoff, C. The system is subject to Newton's third law (symmetry of forces).

The system evolves particles by computing the force to which each particle is subject to, and then integrate them over time using the following linearized equations:

$a = F/m$

$v = v + a * dt$

$p = p + v * dt$

Using these equations, we know that the new position at the end of each time step is given by:

$$x^{new} = x^{old} + dt * v_x$$
$$y^{new} = y^{old} + dt * v_y$$
$$z^{new} = z^{old} + dt * v_z$$

12.3.1 Decomposition

As described in Chapter 1, Charm++ applications are expected to have processor-independent overdecomposition that matches the simulated data and work units. For molecular dynamics, the extreme case of such decomposition is to treat each particle as an independent unit. However, we strongly advice our users to avoid it because of the following reason: representing a particle as an independent unit leads to extremely fine grained tasks/units. As discussed in Section 12.2, this results in high overheads that prevents applications from scaling. In addition, for molecular dynamics, such a choice also results in N_2 communication for finding particle pairs whose distance is less than the cutoff distance.

12.4 Prelude: LiveViz: 2-D Wave Equation on Structured Grid

The wave equation can be stated in the following form,

$$\frac{\partial^2 p}{\partial p^2} = c^2 \nabla^2 p, \tag{12.1}$$

where p is a physical measure such as pressure or the depth of a body of water in a container. This equation can be discretized over a 2d grid using a finite differencing scheme. Assume the pressures are located across a rectangular 2-D grid in x and y dimensions. We call the value of the pressure $p^t_{x,y}$ for time t at location (x, y) in the grid. One solution to the differential equation (12.1) over a grid is

$$p^{t+1}_{x,y} + p^{t-1}_{x,y} - 2p^t_{x,y} = c^2 \left(p^t_{x+1,y} + p^t_{x-1,y} + p^t_{x,y+1} + p^t_{x,y-1} - 4p^t_{x,y} \right). \tag{12.2}$$

The left hand side term $\frac{\partial^2 p}{\partial p^2}$ is represented in terms of the values of p at three consecutive timesteps at grid location (x, y). The right hand side terms are discretized using the p values for the 4 locations adjacent to (x, y) in the grid. One can simply solve for $p^{t+1}_{x,y}$ in equation (12.2) to produce an update rule,

$$p^{t+1}_{x,y} = c^2 \left(p^t_{x+1,y} + p^t_{x-1,y} + p^t_{x,y+1} + p^t_{x,y-1} - 4p^t_{x,y} \right) - p^{t-1}_{x,y} + 2p^t_{x,y}. \tag{12.3}$$

This update rule specifies how the p value for each location in the grid should be computed, using values on the grid from the two previous time steps. This update rule is easy to code

in an application. The application simply maintains two timesteps' worth of p grids, and using these, the next timestep's p grid can be computed. The c term determines the wave speed. This value must be small enough such that the wave cannot move across more than one grid square in a single timestep. Smaller values for c will make the simulation take longer to propagate a wave by a fixed distance, but larger values for c can introduce dispersive and diffusive errors.

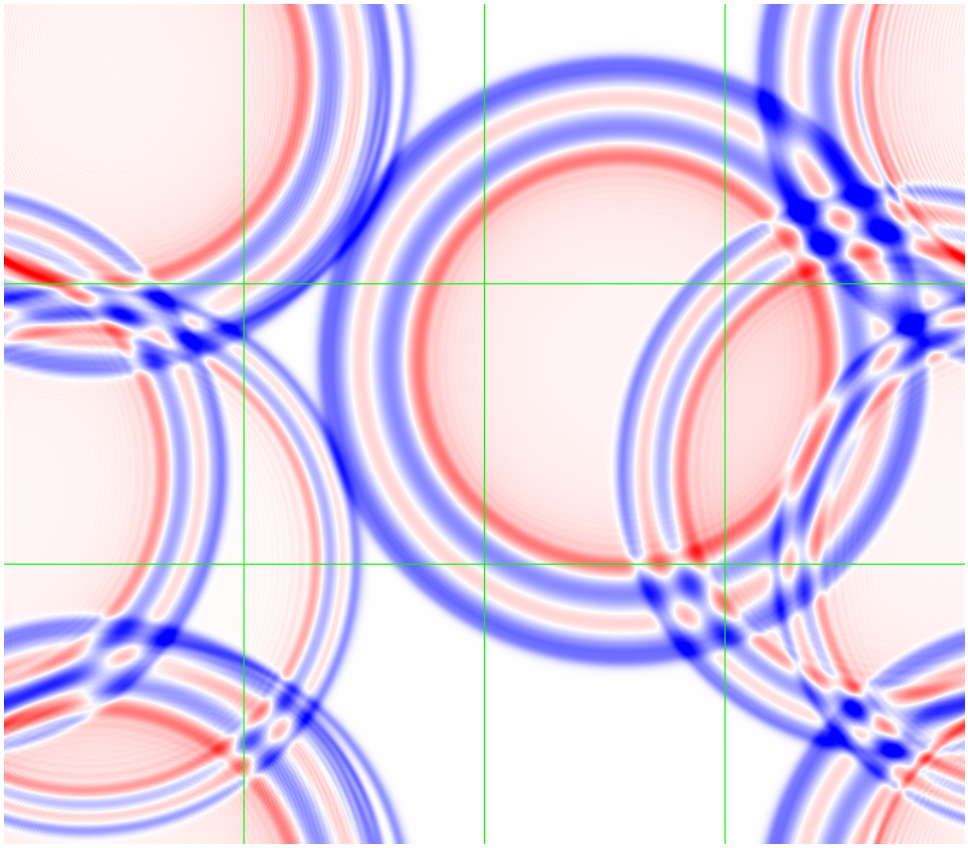

Figure 12.1: A screenshot of the `wave2d` program. Positive values of p on the grid are colored red, while negative values are colored blue. The green lines show the parallel decomposition of the problem onto a 4×3 chare array. The grid domain logically wraps around from left to right and top to bottom.

```
1  mainmodule wave2d {
2
3    readonly CProxy_Main mainProxy;
4    readonly CProxy_Wave arrayProxy;
5
6    mainchare Main {
7      entry Main(CkArgMsg *m);
8      entry void iterationCompleted();
9    };
10
11   array [2D] Wave {
12     entry Wave(void);
13     entry void begin_iteration();
14     entry void recvGhosts(int whichSide, int height, double s[height]);
15
16     // A method for requesting data to be displayed graphically to the user
17     entry void requestNextFrame(liveVizRequestMsg *m);
18   };
19
20
21 };
22
```

Figure 12.2: Wave2D interface file `wave2d.ci`

```
#include "liveViz.h"
#include "wave2d.decl.h"
// Solves the 2-d wave equation over a grid, displaying pretty results through liveViz
// Author: Isaac Dooley 2008

/*readonly*/ CProxy_Main mainProxy;
/*readonly*/ CProxy_Wave arrayProxy;
#define TotalDataWidth  800
#define TotalDataHeight 700
#define chareArrayWidth  4
#define chareArrayHeight  3
#define total_iterations 5000
#define numInitialPertubations 5
#define mod(a,b)  (((a)+b)%b)
enum { left=0, right, up, down };

class Main : public CBase_Main
{
public:
  int iteration, count;

  Main(CkArgMsg* m) {
    iteration = 0;
    count = 0;
    mainProxy = thisProxy; // store the main proxy
    CkPrintf("Running wave2d on %d processors\n", CkNumPes());
    // Create new array of worker chares
    arrayProxy = CProxy_Wave::ckNew(chareArrayWidth, chareArrayHeight);

    // setup liveviz
    CkCallback c(CkIndex_Wave::requestNextFrame(0),arrayProxy);
    liveVizConfig cfg(liveVizConfig::pix_color,true);
    liveVizInit(cfg,arrayProxy,c);

    arrayProxy.begin_iteration(); //Tell all the chares on "arrayProxy" to start
  }

  void iterationCompleted() {   // Each worker calls this method
    count++;
    if(count == chareArrayWidth*chareArrayHeight){
      if (iteration == total_iterations) {
        CkPrintf("Program Done!\n");
        CkExit();
      } else {         // Start the next iteration
        count = 0;
        iteration++;
        if(iteration % 20 == 0) CkPrintf("Completed %d iterations\n", iteration);
        arrayProxy.begin_iteration();
      }
    }
  }
}; // class Main
```

Figure 12.3: Wave2D C++ file `wave2d.C`

```
53
54 class Wave: public CBase_Wave {
55 public:
56   int messages_due;
57   int mywidth;
58   int myheight;
59
60   double *pressure_old;  // time t-1
61   double *pressure; // time t
62   double *pressure_new;  // time t+1
63
64   double *buffers[4];
65
66   // Constructor, initialize values
67   Wave() {
68
69     mywidth=TotalDataWidth / chareArrayWidth;
70     myheight= TotalDataHeight / chareArrayHeight;
71
72     pressure_new  = new double[mywidth*myheight];
73     pressure = new double[mywidth*myheight];
74     pressure_old  = new double[mywidth*myheight];
75
76     buffers[left] = new double[myheight];
77     buffers[right]= new double[myheight];
78     buffers[up]   = new double[mywidth];
79     buffers[down] = new double[mywidth];
80
81     messages_due = 4;
82
83     InitialConditions();
84   }
85
86   Wave(CkMigrateMessage* m) { }
87
88   ~Wave() { }
```

Figure 12.4: Wave2D C++ file `wave2d.C`

```
91  // Setup some Initial pressure pertubations for timesteps t-1 and t
92  void InitialConditions(){
93    srand(0); // Force the same random numbers to be used for each chare array element
94
95    for(int i=0;i<myheight*mywidth;i++)
96      pressure[i] = pressure_old[i] = 0.0;
97
98    for(int s=0; s<numInitialPertubations; s++){
99      // Determine where to place a circle within the interior of the 2-d domain
100     int radius = 20+rand() % 30;
101     int xcenter = radius + rand() % (TotalDataWidth - 2*radius);
102     int ycenter = radius + rand() % (TotalDataHeight - 2*radius);
103     // Draw the circle
104     for(int i=0;i<myheight;i++){
105       for(int j=0; j<mywidth; j++){
106         int globalx = thisIndex.x*mywidth + j; // The coordinate in the global data array (not
107         int globaly = thisIndex.y*myheight + i;
108         double distanceToCenter = sqrt((globalx-xcenter)*(globalx-xcenter) + (globaly-ycenter)*
109         if (distanceToCenter < radius) {
110           double rscaled = (distanceToCenter/radius)*3.0*3.14159/2.0; // ranges from 0 to 3pi/2
111           double t = 700.0 * cos(rscaled) ; // Range won't exceed -700 to 700
112           pressure[i*mywidth+j] = pressure_old[i*mywidth+j] = t;
113         }
114       }
115     }
116   }
117 }
```

Figure 12.5: Wave2D C++ file `wave2d.C`

```
119   void begin_iteration(void) {
120     double *top_edge = &pressure[0];
121     double *bottom_edge = &pressure[(myheight-1)*mywidth];
122
123     double *left_edge = new double[myheight];
124     double *right_edge = new double[myheight];
125     for(int i=0;i<myheight;++i){
126       left_edge[i] = pressure[i*mywidth];
127       right_edge[i] = pressure[i*mywidth + mywidth-1];
128     }
129     // Send my left edge
130     thisProxy(mod(thisIndex.x-1, chareArrayWidth), thisIndex.y).recvGhosts(right, myheig
131     // Send my right edge
132     thisProxy(mod(thisIndex.x+1, chareArrayWidth), thisIndex.y).recvGhosts(left, myheigh
133     // Send my top edge
134     thisProxy(thisIndex.x, mod(thisIndex.y-1, chareArrayHeight)).recvGhosts(down, mywidt
135     // Send my bottom edge
136     thisProxy(thisIndex.x, mod(thisIndex.y+1, chareArrayHeight)).recvGhosts(up, mywidth
137     delete [] right_edge;
138     delete [] left_edge;
139   }
140
141   void recvGhosts(int whichSide, int size, double ghost_values[]) {
142     for(int i=0;i<size;++i)
143       buffers[whichSide][i] = ghost_values[i];
144     check_and_compute();
145   }
146
147   void check_and_compute() {
148     if (--messages_due == 0) {
149       // Compute the new values based on the current and previous step values
150       for(int i=0;i<myheight;++i){
151         for(int j=0;j<mywidth;++j){
152           // Current time's pressures for neighboring array locations
153           double L = (j==0          ? buffers[left][i]  : pressure[i*mywidth+j-1] );
154           double R = (j==mywidth-1  ? buffers[right][i] : pressure[i*mywidth+j+1] );
155           double U = (i==0          ? buffers[up][j]    : pressure[(i-1)*mywidth+j] );
156           double D = (i==myheight-1 ? buffers[down][j]  : pressure[(i+1)*mywidth+j] );
157           // Current time's pressure for this array location
158           double curr = pressure[i*mywidth+j];
159           // Previous time's pressure for this array location
160           double old  = pressure_old[i*mywidth+j];
161           // Compute the future time's pressure for this array location
162           pressure_new[i*mywidth+j] = 0.4*0.4*(L+R+U+D - 4.0*curr)-old+2.0*curr;
163         }
164       }
165       // Advance to next step by shifting the data back one step in time
166       double *tmp = pressure_old;
167       pressure_old = pressure;
168       pressure = pressure_new;
169       pressure_new = tmp;
170       messages_due = 4;
171       mainProxy.iterationCompleted();
172     }
173   }
```

```
175   // provide my portion of the image to the graphical liveViz client
176   void requestNextFrame(liveVizRequestMsg *m){
177
178     // Draw my part of the image, plus a nice 1px border along my right/bottom boundary
179     int sx=thisIndex.x*mywidth; // where my portion of the image is located
180     int sy=thisIndex.y*myheight;
181     int w=mywidth; // Size of my rectangular portion of the image
182     int h=myheight;
183
184     // set the output pixel values for my rectangle
185     // Each RGB component is a char which can have 256 possible values.
186     unsigned char *intensity= new unsigned char[3*w*h];
187     for(int i=0;i<myheight;++i){
188       for(int j=0;j<mywidth;++j){
189
190         double p = pressure[i*mywidth+j];
191         if(p > 255.0) p = 255.0;    // Keep values in valid range
192         if(p < -255.0) p = -255.0;  // Keep values in valid range
193
194         if(p > 0) { // Positive values are red
195           intensity[3*(i*w+j)+0] = 255; // RED component
196           intensity[3*(i*w+j)+1] = 255-p; // GREEN component
197           intensity[3*(i*w+j)+2] = 255-p; // BLUE component
198         } else { // Negative values are blue
199           intensity[3*(i*w+j)+0] = 255+p; // RED component
200           intensity[3*(i*w+j)+1] = 255+p; // GREEN component
201           intensity[3*(i*w+j)+2] = 255; // BLUE component
202         }
203
204       }
205     }
206
207     // Draw a green border on right and bottom of this chare array's pixel buffer.
208     // This will overwrite some pressure values at these pixels.
209     for(int i=0;i<h;++i){
210       intensity[3*(i*w+w-1)+0] = 0;     // RED component
211       intensity[3*(i*w+w-1)+1] = 255;   // GREEN component
212       intensity[3*(i*w+w-1)+2] = 0;     // BLUE component
213     }
214     for(int i=0;i<w;++i){
215       intensity[3*((h-1)*w+i)+0] = 0;   // RED component
216       intensity[3*((h-1)*w+i)+1] = 255; // GREEN component
217       intensity[3*((h-1)*w+i)+2] = 0;   // BLUE component
218     }
219
220     liveVizDeposit(m, sx,sy, w,h, intensity, this);
221     delete[] intensity;
222
223   }
224
225 };
226
227 #include "wave2d.def.h"
```

Chapter 13

Advanced Topics

New organization:

13.1 Modules

13.2 CkLoop

13.3 Entry method attr

13.4 Tracing options

13.5 Adv LDB

13.6 Dynamic Insertion

Chapter 14

Performance Analysis with Projections

Performance analysis is very important, and relatively complex, in parallel programming. After all, the main reason for writing a *parallel* program is to run computation faster. Except for a small subset of "simple" applications, it is often pretty tricky to get good performance for your application. As such, even after having chosen a good parallel algorithm, and writing what you think is a good parallel implementation, parallel programmers often find themselves staring at applications that didn't perform as well as they expected. The natural next step in such scenarios is to collect more information about the application execution, and find the reasons for poor performance.

As stated in earlier chapters, the runtime system in Charm++ both schedules computation and mediates communication. Hence, it can collect a large amount of information that is useful for analyzing performance. To enable it, the user should provide the following link time option when building the executable: -tracemode projection. Note that this option only works if Charm++ was built to allow tracing. This can be done by providing –enable-tracing build time option when Charm++ is compiled. The traces, called Projections logs, collected by the Charm++ RTS are written to files at the end of an execution, generally in compressed format. Many options are available to control the output of the trace files; please refer to the manual entry here for more details.

Projections is a java-based tool used for visualizing and analyzing the data in the trace files. It can be downloaded either via git using the following command, or from the software section of the Charm++ website.

```
git clone http://charm.cs.illinois.edu/gerrit/projections
```

To build Projections, you would also need a working java runtime environment (JRE version > 1.5), accessible in your default path, i.e. commands such as java should be in your

default path. Running a simple make in the Projections' directory will compile and write the binaries in the bin directory.

14.1 Loading logs and Range selection

To run Projections, in a shell, type the following

```
projections <yourFile.sts>
```

The sts file is one of the files created along with the trace files. It contains metadata about the Projections logs. If a sts file is not specified when Projections is launched, it can be selected via the GUI's file dialog File -> Open File(s). When the sts file is loaded, the user can choose a View in the Tools dialogue to analyze performance. We divide the views into three categories: Aggregated analysis, Individual analysis, and Derived Analysis.

Before we provide details on these views, let us first look at the how one can select range of various entities in these views. Figure 14.1 shows a dialogue box for selecting range in Projections. The first and the third line in the box tells the user the valid range for processors and time, respectively. The user can specify the processor range in Line 2. This range can be a comma separated list of subranges; each subrange is specified using at most three numbers in $L - U[: S]$ format. Here L is the lowest value in the range, while U is the highest value. The argument S is optional, and specifies the step; by default, S is 1. For example, the given range in the figure $0 - 2, 4 - 7 : 2$ equates to processors $0, 1, 2$ from its first subrange and processors $4, 6$ from its second subrange.

In Line 4, the time range, for which a view should be loaded, is specified. The default unit of the time entries is milliseconds (ms); users can use seconds (s) as an alternate unit. A selected processor and time range can be saved in memory and in disk for later use using the buttons in the next two lines. This is extremely useful when performing complex analysis where multiple ranges and multiple views are being compared, often over days by a group of people. Finally, the last section of the dialogue box allows the user to make view specific choices.

14.2 Aggregate Views

As suggested by the name, the Aggregate views show the data added across the given range of processors and/or time. As such, they are useful in performing initial high-level analysis.

14.2.1 Time Profile

Time Profile is one of the most commonly used view that aggregates data across processors. As shown in Figure 14.2, this view shows the amount of time spent in various entry methods

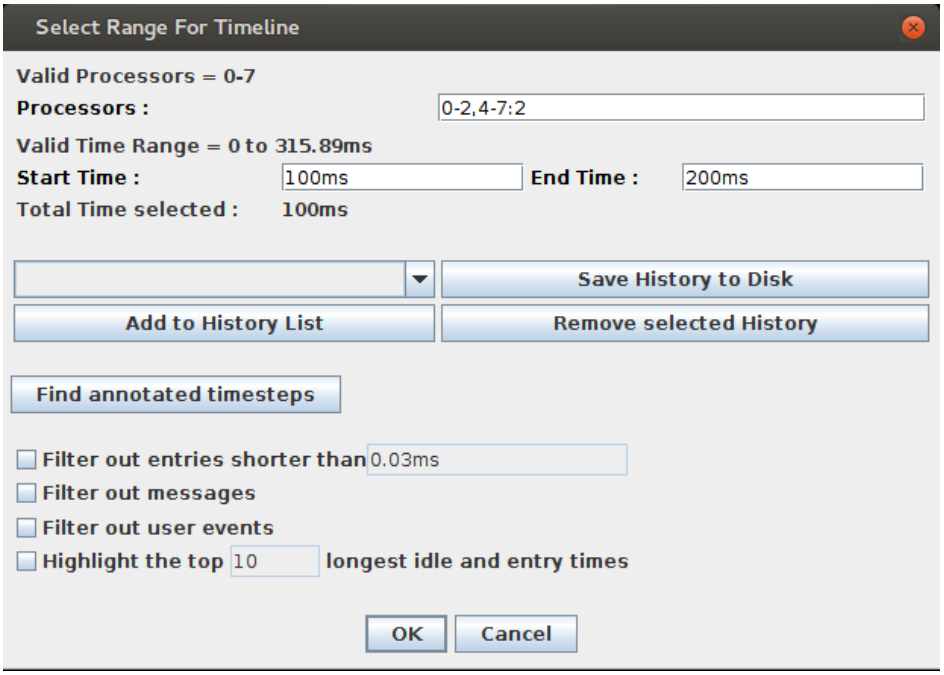

Figure 14.1: Range selection box: ranges can be chosen and saved to disk for future use.

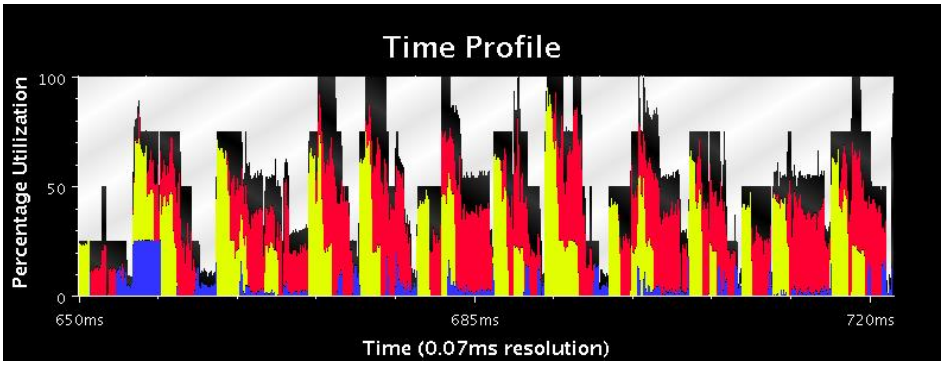

Figure 14.2: Time Profile view: x-axis is execution progress in time; y-axis is time spent in entry methods aggregated across all processors.

aggregated across all processors. Shown in a stacked manner by default, each colored area stands for an entry method. By hovering the mouse over a given region, one can view the entry method's name and the chare class to which it belongs. One could use the axis-scale buttons to zoom in/out on regions of interest. By default, the white colored area represents idle time, while the black region shows the overheads of the runtime system. The mapping of entry method to color could be changed via the color box (Color Scheme -> Choose Entry Colors).

The Time Profile graph provides a lot of information about the overall execution. First, it tells the users about the amount of idle time and RTS overhead observed during the execution. This can help the user judge if the system resources are being efficiently utilized. Next, this view helps users estimate the dominant entry methods and the associated computation. This information can be used to further target both parallel and sequential optimizations of parts of the code. Finally, using Time Profile, the user can understand how the behavior of the code evolves with time as the execution progresses.

14.2.2 Usage Profile

In contrast to across processor aggregation in Time Profile, the Usage Profile graph presents aggregated information over time for individual processors. In this view, as shown in Figure 14.3, one can view time spent in various entry methods by each processor in the form of a bar graph. In this view, X-axis is the processor number, while the Y-axis is the percentage time spent in an entry method (stacked). Each color, as expected, represents an entry method. Note that the first bar in the graph shows the average percentage time spent in entry methods across processors. Mousing over each colored rectangle, you can see additional details, such as the name of the entry method, the number of calls, and total time spent in them, etc.

This view's utility lies in the comparison it provides among processors. The user gets to compare how different processors spend their time, and find anomalies if they exist. Load imbalance and other load-variation patterns are also easily observable in this view.

Of course, when the number of processors is very large, this view becomes difficult to deal with. Imagine 4000 bars for 4000 processors (let alone 500,000 that many Charm++ applications have scaled to). We will see other views that help you scale, later in this chapter.

14.2.3 Timeline

This is probably the most information-rich view in projections, and one thats very useful for detailed, focused, analysis of performance. In this view (see Figure ??? NEED ONE), the X-axis is time. Each horizontal line of blocks represent happenings on a PE organized in time-order. The main components are rectangles that show individual entry method executions. Idle time is indicated by a different custom color (typically white) and overhead (i.e. the time

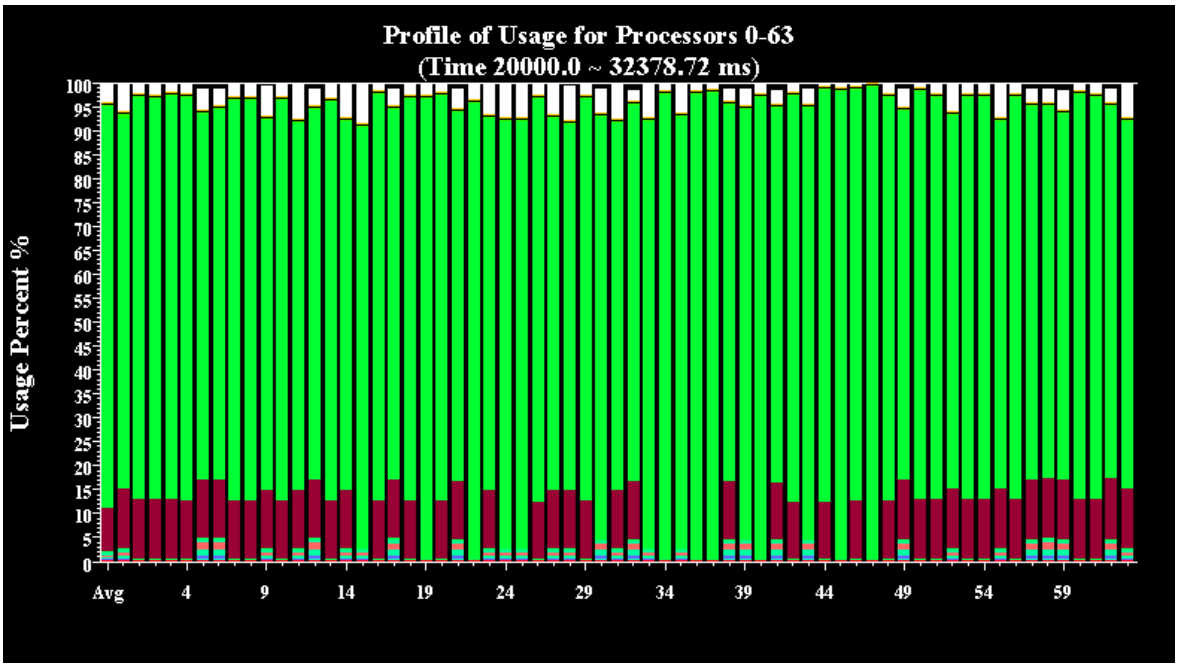

Figure 14.3: Usage Profile: view to compare utilization across processors.

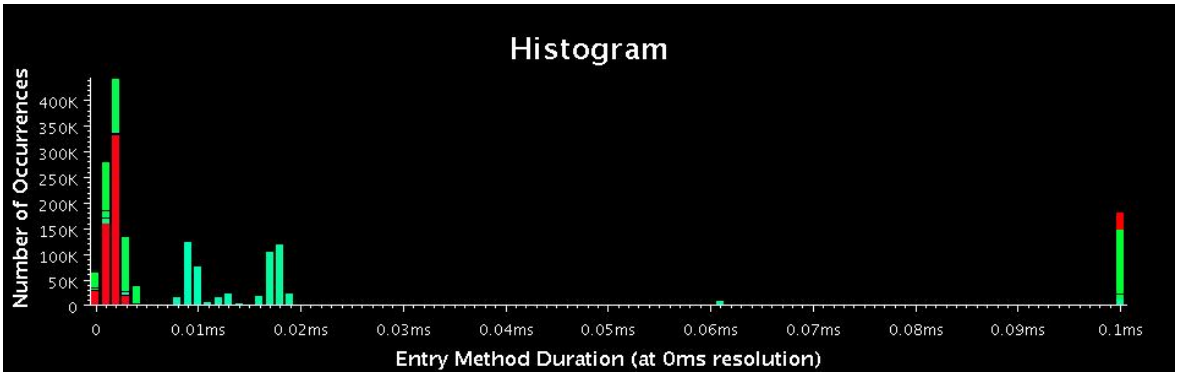

Figure 14.4: Histogram: time spent in various entry methods shown in frequency domain.

spent in the runtime system, including that for receiving and copying messages) is shown in black. Mouse-over for the entry method rectangles shows a rich set of data, including size of the incoming message that triggered it, the number of messages in sent (with their time, type, size, and destinations when known), etc. You can check predecessor chains (which entry methods were on the critical path to this one), and successor entry method instances.

In typical use, only a small number of processors are shown in the timeline. You can make compact forms, which lose some details but accomodate more processors in a screenful). It is worth noting that you can insert timelines for processors in this view, by clicking them from other views, such as the outlier view we describe next.

14.2.4 Outliers

When you are analyzing performance on a run with a large number of processors, over a phase of the program, you cannot visually inspect all processors to make a performance diagnosis. It is helpful to identify a subset of "interesting" processors. The outlier view comes in handy for this purpose.

You can identify outliners among many different aspects. A popular one is "idle time". A typical dialog will specify "please find 20 least idle processors in the time-range from 5.32 to 5.44 seconds". The *least idle* corresponds to most overloaded processors, and this helps idenify the reasons of load imbalances. The view shows bar charts for the most loaded processors, along with the average bar-charts. Work items that you did not expect to take as much time, but did, can be identified this way. A click on one of the bars loads (or inserts) it in the time-line view, so you can view what that processor was doing for the given duration in great detail.

In addition to idle time, you may select other aspects such a outgoing message volume, incoming message volme (CAN WE?? WE SHOULD), and WHAT ELSE?

14.2.5 histogram view

This view is another scalable view. I.e. it comes handy when you are dealing with a large number of processors.

14.2.6 communication over time

14.2.7 communication acrossprcessors

———————- NEED TO INSERT EXAMPLES OF THESE VIEWS

SHOULD WE PROVIDE A PERFORMANCE ANALYSIS CASE STUDY THAT SHOWS THE USE OF MULTIPLE VIEWS?