A Tutorial Introduction to Charm++

Laxmikant Kale
Parallel Programming Laboratory
Department of Computer Science University of Illinois at Urbana Champaign

November 16, 2003
# Contents

1 Introduction .................................................. 5

2 Overview of Concepts ......................................... 9
   2.1 Central Ideas ............................................. 9
      2.1.1 Virtualization: .................................... 9
      2.1.2 Asynchronous Method Invocation .................. 11
      2.1.3 Consequence: Message Driven Execution ........ 11
   2.2 Entities in Charm ......................................... 12
      2.2.1 Chares .............................................. 12
      2.2.2 Grouping Chares Together ......................... 13
      2.2.3 Information Sharing abstractions ................. 14
   2.3 Examples .................................................. 15
      2.3.1 N non-attacking queens ............................ 15
      2.3.2 Conways’s Game of Life ........................... 16
   2.4 Notes ..................................................... 17

3 Basic Charm Constructs through examples .................. 19
   3.1 Usage of Main Chare ..................................... 19
   3.2 Array Hello .............................................. 21
   3.3 prime .................................................. 23
   3.4 array ring .............................................. 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Parallel Prefix</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Jacobi</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>Migration</td>
<td>28</td>
</tr>
<tr>
<td>3.8</td>
<td>Load imbalance</td>
<td>28</td>
</tr>
<tr>
<td>3.9</td>
<td>Load balancing</td>
<td>28</td>
</tr>
<tr>
<td>3.10</td>
<td>Structured Dagger</td>
<td>28</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Since the advent of digital computers half a century ago, computer users have always hungered for faster computers. More precisely, they have been amazed at the speeds of the available computers, figured out ways of using that speed to solve their problems faster, to find better solutions, to enlarge the scope of solvable problems, and then hungered after more powerful computers to solve the new problems faster.

The demand for more power seems insatiable: when new computers are introduced, they seem so powerful compared to their previous generation that opinions like ”Who needs any more power than this?” seem to dominate, soon to be displaced because of new applications and new levels of performance expectations. Thus when the Cray supercomputers were introduced in 1976, a prominent computer scientist and industry insider stated that the demand for these will be less than 10 per year. This obviously turned out to be wrong.

Such demands for computational power were continuously met by rapid advances in hardware technology and microprocessor architectures. This in turn fueled the demand for even more powerful computers. Together, these technological advances led to the phenomenon of shrinking prices with dramatically increased performance. Thus the Pentium-based PCs sitting on desks in homes are millions of times faster than the earliest computers, yet thousands of times cheaper than them.

Most of these improvements have come from the ability to produce smaller electronic circuits on silicon chips. The smaller you can make the circuit, the faster it is and the cheaper it is to produce. However, this happy trend cannot be continued forever. Physical limitations and the increased probability of failures will put a damper, if not a ceiling, on how far the trend can go.

Parallel computing is a way out of this problem that will continue the trend in speed improvements for years to come. The idea here is simple: to solve a com-
putational problem faster, use many processors to work on different parts of the problem. However, anyone who has tried reduce task completion time 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 around 1985, many commercial parallel computers became available. In 1994, there are computers available with tens, hundreds, or even thousands of processors each. Most of them employ state of the art microprocessors, each capable of hundreds of millions of operations per second.

There are many applications that can benefit from such machines to date. These include large applications such as global weather forecasting that can use thousands of processors and small applications such as text processing that can benefit from speed-ups offered by a few processors. There are emerging applications in multimedia, online transaction processing, decision support systems that “mine” and analyze huge amounts of data and so on. And surely, you, the readers, will invent newer applications that no one has thought of before, in the coming years.

In this book, you will learn about a new and effective way of coordinating these parallel computers — a new methodology for parallel programming. This methodology is embodied in a parallel programming system called Charm. One of the basic premises on which our methodology is based is: although parallel computers have a variety of different architectures, a simple cost model unifies them all and 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 is sitting in a processor’s cache or its private memory. Remote data is all other data — on some machines this may be data in other machines private memory while on others it may be data in global shared memory modules. A programming language that exposes this cost model and encourages writing of programs that respect locality of data access is likely to lead to efficient programs. As you will see in subsequent chapters, Charm indeed is such a language.

The other important idea embodied in Charm is that of an optimum division of labor between the programmer and the “system”.

One thing to keep in mind is that Charm is not just a message passing layer or a portability layer. It represents a higher level methodology for developing parallel programs and provides the compiler and run-time support necessary for this.

Charm is a coordination language in the sense that its constructs specify only the parallel aspects of a program. Of course, any parallel program also includes many sequential components. Charm itself does not take any position on how those should be expressed. In particular, the sequential components can be expressed
using popular languages like C, C++, and Fortran. Practical implementations of Charm in each of these languages exist and are called C-Charm, Charm++, and FortCharm respectively. In addition, the central programming ideas (cost model and virtualization) of Charm have been embodied into AMPI (Adaptive MPI), which supports MPI's coordination mechanisms (messages) for virtualized MPI processes. In this book, we will use examples from these languages interchangeably, but mostly will focus on Charm++.

Before we plunge into the details of the language, let us overview the basic principles it incorporates.
Chapter 2

Overview of Concepts

This chapter will provide an overview of the Charm language and its associated tools. The idea is to provide you with a broader perspective before we start learning each concept in detail.

2.1 Central Ideas

Charm++ is a parallel programming approach that combines two important ideas: Virtualization and Asynchronous method invocation.

2.1.1 Virtualization:

When one writes a parallel application, one has to decide

1. How to decompose (or partition) the data and computations,
2. How to map the resultant partitions and computations to processors,
3. In what sequence to execute multiple computations assigned to each processor.

One then also needs to express these decisions using the specific constructs provided by the machine. However, this step has become simple, if you are using a universally supported interface such as MPI.

In traditional explicit parallel programming approaches for distributed memory machines, such as MPI, all three decisions must be taken by the programmer. This makes writing parallel programs quite complex, especially when the program exhibits dynamic behavior. Parallelizing compilers aim at automating all 3 sets of
decisions, but are often unsuccessful in extracting adequate parallelism from the sequential program.

Figure 2.1: Virtualization: User’s view of the program does not involve “processors”.

The idea of virtualization, as embodied in Charm, is to automate steps 2 and 3, while leaving decomposition to the programmer. Thus, the programmer divides the computation into a large number of “chunks” (also called virtual processors) which are then assigned by the runtime system to processors. The runtime system is then also free to reassign them during program execution as it sees fit (See Figure 2.1). Since the application program is written in terms of interactions between virtual processors, without any reference to real processors, user code doesn’t change when such reassignments are done. Of course the runtime system now has to keep track of where (i.e. on which real processor) these virtual processors reside, so it can deliver data to them efficiently.

The virtual processors themselves need to exchange data and coordinate among themselves. Charm++ provides one particular coordination method described below. However, other languages in the Charm family use other coordination mechanisms (for example: in AMPI, virtual processors communicate via messages, with MPI syntax). What is common to all these languages is the orthogonal notion of virtualization.

Virtualization leads to better software engineering, since entities are not clubbed together simply because they are on the same processor. It also leads to automatic adaptive overlap between communication and computation, and encourages modularity. ([?]). Most importantly, it gives the runtime system a degree of freedom that permits runtime optimization based on observations and measurement made at runtime.

2.1.2 Asynchronous Method Invocation

When an object, say A, invokes an entry method in another object, say B, we don’t want A to be stuck waiting for the invocation to travel to the possible remote processor where B resides, let that processor schedule the invocation, and then the return value to be passed back to it. For this reason, invocation of entry functions in Charm are non-blocking, or asynchronous: that means an entry function invocation is just a one-way call. You send the data to the remote object, but you don’t wait for it to give you a value back. (That is, no value is returned to you.) If you want the remote object to send some "result" back to you, this can be done by having the remote object asynchronously invoke another entry function of your object. An analogy may help here. A public or private function call is like a telephone call: you call the other party, ask them a question, wait for a reply, and then continue with
your work. Invocation of entry functions is, on the other hand, is like a mail message: you give the other party some information and possibly instructions, and they look at it and act on it at their leisure. They might even collect more information by phone or by mail before they respond to you. They might even delegate the job of responding to you to someone else. In some cases, you may not need any response — you are just providing them some information which they might store or act on themselves. All of these modes of behavior are possible with the asynchronous invocation of entry functions. Indeed, such an invocation can be thought of as sending a message to the object (We may use these terms in the rest of the book interchangeably).

2.1.3 Consequence: Message Driven Execution

Charm allows multiple objects per processor. It is not uncommon to have programs in which there are hundreds or even thousands of objects per processor. It is possible then, that an individual processor might have a number of “messages” corresponding to asynchronous invocations of its objects pending at any given time. These messages are kept in a pool or queue. There is a separate pool on each processor. (See Figure ??). The processor scheduler picks a message from this queue, identifies the object to which it is directed and the method within the object that must be used to handle it, and invokes this method with the data in the selected message as its parameters. When the method (entry function) finishes execution the control comes back to the scheduler. The scheduler then picks another message and schedules its execution.

This style of execution is called message driven execution. It is one of the major features of Charm that gives Charm programs a significant performance advantage. Message driven execution insures that a processor does not remain idle unnecessarily. When one object is waiting for some data — which will eventually come in a message from another object — another object can use the processor; also, a single object can wait for multiple possible message simultaneously. Whichever message arrives first will get the attention of the object. This gives Charm programs a degree of adaptiveness to runtime conditions.

2.2 Entities in Charm

A Charm++ program consists of Chares: or data-driven objects, Chare-Arrays and Information sharing abstractions. In addition, a low-level construct called “object-groups” is also provided, which is used by some library developers, and should be considered an advanced feature.
2.2.1 Chares

Respect for data locality is essential for design of efficient parallel programs. The notion of locality is captured nicely in the idea of an object. An object, in essence, consists of some data and a collection of procedures that can update this data. The object encapsulates its data in the sense that the data can be accessed or modified only through the functions provided by the object. These functions are also called "methods" of the object. In the parallel programming context, objects also express locality. Any data that is within the object can be assumed to be local data that can be accessed with speed. Data from another object may or may not be accessible quickly depending on whether that object resides locally or not.

To make the programmer aware of the potential cost of interacting with remote objects, we must distinguish calls to methods in local objects from calls to methods of remote objects. For this reason, a Charm program consists of a special form of object called a chare. A chare consists of some local data, private methods that can be called only from within other functions in the same chare, and entry methods that can be called by any object.

You can create new chares anytime and from any other chare in your 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 you make the call to create the object. That call immediately returns after depositing a seed for the new chare with the system. All that you are guaranteed is that eventually, on some processor, that chare will be created and will execute its initial message provided by you. In fact, all system calls in Charm are "non-blocking." They do not wait for actions on remote processors. To send a message to an existing chare, you need its ID. A chare can find its own ID by making a system call and pass this ID to other chares. This can be done in the creation message, for example. For example, a chare X may send its ID in the creation message when it creates another chare Y. Y can then send its ID to X in another message, establishing communication between the two. Other ways of communicating this ID will be discussed later.

At a point in time there may be many messages in the pool of messages. Charm provides a default selection strategy (which is fifo) for picking the next message for execution. You can replace this default strategy with any of the available strategies that Charm supports. We will learn about the appropriateness of specific strategies in later chapters.
2.2.2 Grouping Chares Together

Charm also provides two constructs that group a bunch of objects together: Chare arrays and Object groups.

Chare Arrays

A chare array is a collection of chares with a global name for the collection. There may be multiple chare arrays in a single application. Each element of the collection is identified by an index. The index may be something simple like an integer, but can be any general user-defined type. The name “chare arrays” derives from a common use: a collection of chares indexed by a single integer. In such a usage, all the indices of a chare-array may form a contiguous set of numbers. So, for example, you may have a chare array represented by its global id A, and it may have members identified by indices 0..99. However, even within one-dimensional arrays, you can have chare arrays with non-contiguous indices (e.g. an array consisting of 1000 elements with indices ranging between 1 billion and 2 billion). Other examples of index types include 2-D, 3-D and higher dimensional integer indices, bit-vector indices representing a position in an oct-tree etc.

It is important to remember that a chare array does not necessarily represent an array of basic types. (It is not like an hpf array, for example). Each element of an array is a message driven object, which may hold whatever data the application may choose to assign to it — linked lists of particles, arrays of grid points, finite-element subgrids, or trees, for example).

A program may invoke an entry method on an individual element of a chare array, or broadcast a method invocation to all its members. The chare array elements may also participate in reduction operations (e.g. adding up data across all its members). The important feature of chare-arrays is that the elements of an array may be migrated across the processors by the runtime system. Yet the user program does not need to be aware of where each array element is located. You can invoke a method on “element numbered 8”, and the system will find where this elements lives, and deliver the method invocation to it, even when the element has migrated to a different processor recently.

Chare Groups

An object group, also known as ”Branch Office Chare” (BOC), consists of exactly one representative chare on each processor. Each representative or “branch” ”answers to” the same name. Therefore, a regular chare that has landed on some processor under the control of the dynamic load balancing strategy can still make a function
call to the local branch of a BOC without having to now which processor it is on. It is like opening your eyes in an unknown town and picking up the telephone to call the local branch of Domino’s Pizza. One can broadcast to all branches of a BOC or send a message to a branch of the BOC on a specific processor, in addition to invoking its member functions on the local branch. As we will learn later, a BOC is a versatile construct that is useful for static load balancing, local utilities, distributed services, and distributed data structures.

In current usage, BOCs are used almost exclusively by library writers and system programmers in Charm++. They should be avoided (in favor of object arrays) by application programmers.

2.2.3 Information Sharing abstractions

So far, we have instances of chares chare-array elements that communicate with each other via messages. But a message is a very narrow mechanism for exchanging information in a parallel program. It is useful when the creator of the information knows the identity of its consumer. A general mechanism such as a shared variable, on the other hand, is too amorphous — no one needs the full generality of a shared variable that any object can update and access, and this generality is quite expensive to support. Charm provides multiple information sharing abstractions for this purpose. The six abstractions supported by Charm, in addition to asynchronous method invocations, cover most of the information sharing needs efficiently. These abstractions may be implemented differently by the Charm runtime system on different machines, but the Charm programmer isn’t aware of the implementation and simply uses the abstraction presented to them.

The simplest and most commonly used information-sharing abstraction is a read-only variables. Such variables are assigned values at the beginning of the computation, and never change their values during computation. We will introduce the other abstractions (write-once variables, accumulators, monotonic variables, and distributed tables) in later chapters as needed.

So a Charm computation consists of chares, chare-arrays and branch office chares that send messages to each other (i.e. invoke methods on each other asynchronously), and dynamically create new chares, chare-arrays and BOCs. These execute under the control of a message-driven scheduler that invokes appropriate methods in chares or BOCs to handle these messages. In addition to sending method-invocations to each other, they also exchange data using specific information sharing abstractions.
2.3  Examples

Before we describe how these constructs are expressed concretely, with syntax and boring details, let us look at a couple of examples to understand how these constructs help in writing parallel programs. The examples we chose are toy problems, intentionally simple so we can focus on the expression of parallelism.

2.3.1  N non-attacking queens

Consider the problem of finding all solution to the N — queens problem: on a NxN chess board, place N queens such that no queen attacks another. A queen is said to attack another queen if the two are in the same row, same column, or same diagonal.

In a simple formulation for solving this problem, one starts from an empty board and then systematically enumerates all possible ways of placing a queen on the first row, then for each such placement, enumerating all possible ways of placing a queen on the second row and so on. The structure of the computation then resembles a tree. Of course, you don’t have to continue the enumeration beneath a node in the tree if you have already detected a conflict — and you shouldn’t, because otherwise you will search an unnecessarily large tree. This pruning rule, although helpful, makes the tree irregular. Among the children generated by a node, one cannot predict how big a search tree each will generate, some might be pruned right away, while others might give rise to large searches. Such irregularity makes this problem not amiable to static distribution using BOCs. We need the power of dynamically load balanced chares. A chare based formulation will assign one chare for each node in the tree. A chare is given a partially filled board in its creation message. It then identifies all of the safe places in the next row where a queen could be placed and fires a new chare for each of the new feasible positions generated. The new chares are dynamically balanced by the system — i.e. the system takes a responsibility for assigning them to processors in such a way to balance load across processors.

When a chare finds that the board it got is already filled with N queens, it reports the solution. This reporting can be done by simply printing the solution. For larger values of N, the number of solutions may be too large to print, and one may be interested in only the total number of solutions. Each chare that finds a solution needs to add to the count of solutions found and some chare in the system eventually needs to find the total and print it. This form of information sharing is supported by the accumulator abstraction in Charm which can be used to obtain this count efficiently.

One needs to know when the search has ended. This can be found by using a system library that detects quiescence as explained later.
2.3.2 Conways’s Game of Life

The Game of Life involves a NxN grid. At each cell on the grid, either there is “life” or there is none. At every time-step the system evolves using the following rules:

1. Birth: If there is an empty cell such that among its 8 neighbors, 3 of them have life, then a new entity is born at this cell.

2. If a live cell has fewer than ?? neighbors that have life, it dies.

To study the evolution of the system in parallel using Charm++, we divide the NxN board into multiple pieces each of size at most kxk. The pieces together constitute a 2-dimensional array of chares. Since you need to know your neighbor’s data for each cell, each chare sends its border to the 4 neighboring chares, and copies data sent by them into a ghost region around its array. It then updates its cell in accordance with the rules above.

Charm’s dynamic load balancing can useful here, if you optimize your program so that a kxk cell that is empty and has empty borders, does not do any cell-by-cell computation. Some chares will be idle, while others have O($k^2$) work to do. Charm++ can automatically balance the load by moving chares around as the life-board evolves.

In a more general formulation of the game of life, the board size is not fixed. As the life expands outward, you have to expand the board. This also can be easily accomplished in Charm++, since new elements can be inserted in the object array representing the board at any index chosen.

2.4 Notes

We are taking some liberties in the interest of simplicity while describing technical concepts in this book. After all, this is a textbook a not a technical paper. Whenever possible, we will explain the subtleties involved in the "Notes" section of each chapter. As an example, we said that objects automatically induce the expression of locality. This is true only if we assume that each object resides completely on a processor or at least that its data is stored "locally accessible" memory modules. Distributed objects, for example, would violate this assumption. For this reason, we think of branch office objects as an ordered set of objects although they can be thought of as distributed objects.
Chapter 3

Basic Charm Constructs through examples

The best way to learn Charm++ is to follow a series of program examples, each of which introduces some new concepts. You should also write a few example programs based on these concepts as you learn them.

3.1 Usage of Main Chare

We will begin by writing an extremely simple program which prints "Hello World". Even in this simple program, we get to introduce several features of Charm++. We will see what an interface file is, and how to define it. We will see what special code is needed to exit a Charm program, and why. We will see how to compile and run a Charm++ program.

The execution of every Charm++ program begins with the creation of one instance of every chare class designed as a mainchare, and execution of its constructor with (CkArgMsg*) as its sole argument. Our simple Charm++ program has a single chare class (which happens to be named “main”) which is the mainchare.

To tell the runtime system which (of possibly many) class(es) are designed as mainchare(s), we use a special interface file. The interface file has other purposes as well, which will be illustrated as we look at more complex examples.

The interface file for this example is shown in Figure ??.

The first word in this file is either “module” or “mainmodule”. By using the keyword mainmodule, we inform the system that this module contains a main chare.
“hello” is the name of the module. “main” happens to be the name of the chare being defined here. Although it is designed as a mainchare, its name does not have to be main. The main chare has only one “entry” method that the system needs to know about, and that is the constructor method. A chare class may have many methods, and only these methods that may be invoked asynchronously (and remotely) need to be specified in the interface file.

The parser for .ci files is finicky, and it needs the semicolons after every class and entry method specification.

The .C file for our “Hello World” program is shown in Figure 3.1.

In the constructor of the mainchare “Hello World” is output, using a ckout class. Charm++ provides both C and C++ style methods for doing terminal I/O. In place of C-style printf and scanf, Charm++ provides CkPrintf and CkScanf. For C++ style stream-based I/O, Charm++ offers ckout and ckerr in the place of
cout and cerr.

Note the constructor \texttt{main(CkMigrateMessage * m)}. This is a migration constructor which is normally empty and has to be written for any chare implemented. The significance of this constructor will be discussed later. For the proper termination of the Charm++ program, a \texttt{CkExit} call should be made. This is an indication to the Charm Kernel that there are no more messages to be processed. Without this call, the program will not terminate.

Why is this call needed? Remember the earlier discussion of message driven execution in Section 2.1.3? There are several objects on each processor and a message-driven scheduler is needed that repeatedly selects the next message from the queue of waiting messages, and executes the methods indicated by the message. Can this scheduler stop (and terminate the program) if the queue of messages is empty? This won’t work on multiple processors, because even if the queue on processor A is empty, processor B may be humming along, and some object there will send a message to an object on A soon. So, the system cannot shut things down on its own. Instead the programmer is required to call \texttt{CkExit()}. If \texttt{CkExit} is called on any one processor, the system broadcasts that information to all the processors, and all the schedulers will stop and exit after they have finished executing the current method that they are already running.

3.2 Array Hello

Now, let us write a simple parallel program. We will create \(N\) virtual processors. Each one of them holds one number. (For simplicity, let us say that the processors are numbered contiguously and that \(i^{\text{th}}\) processor holds the number \(i^2\)). We want them to carry out a simple batton-passing algorithm so that \(i^{\text{th}}\) virtual processor has the sum of all numbers on virtual processors with serial numbers less than or equal to \(i\).

We will use (and introduce) 1-dimensional chare arrays to implement the virtual processors. We will learn about how a chare-array-element can find its index. We will also learn and use asynchronous method invocation.

The overall idea for the program is: we will have a main chare that will read \(N\) from command line and create a 1-D chare-array with \(N\) elements with indices 0..\(N-1\). The main chare will send a message to the 0’th element. Each element, when it gets the message, will add its contribution and send a new message (method invocation) to the next element. It will also print the sum that it sent rightwards.

The interface file for this program is shown in Figure 3.2. It specifies that the module will have two chare classes: one main chare called \texttt{main}, and a chare array
element class called A. A has a hello method in addition to its constructor.

The header file is shown in Figure 3.2.

The first line includes hello.decl.h file. Where did this file come from and what does it contain? Remember the idea of proxy class we discussed earlier in Chapter ref?? . from the interface definitions provided in the “.ci” file the charm translator produces additional classes for proxies. The declarations of these classes are stored in the “.decl.h” file. The first part of the name of the file (in this case “hello”) is the module name specified in the interface (.ci) file. Although it is customary to name the module the same as the file name (so, in this case, hello.* are the file names, and “hello” is the module name), it is not required. If the module were called helloMod, the file to include will be: “helloMod.decl.h”.

The method definitions for the classes declared in this file are stored by the charm translator in a file named hello.defs.h (in general MODULENAME.defs.h).

This file has to be linked in, but it doesn’t have to be included. It is customary to include in at the end of the “.C” file.

Charm requires you not use global variables that you can change during the execution of programs. All the changeable data must be inside some chare objects. However, it is convenient to have global variables that do not change their value after it is set initially. The readonly variables of Charm++ satisfy this need. The ROs must be set only during the execution of the constructor of a main chare. The system will copy those values to all the processors, where they can be accessed just
#include "hello.decl.h"

/*readonly*/ CProxy_main mainProxy;
/*readonly*/ int nElements;

/* main chare */
class main : public Chare {
public:
    main(CkMigrateMessage *m) {}
    main(CkArgMsg *m);
    void done(void);
};

class Hello : public ArrayElement1D {
public:
    Hello(CkMigrateMessage *m){}
    Hello();
    void sayHi(int hiNo);
};
like regular global variables. To enable the system to know which values to copy, it is necessary to declare all the readonly variables in the interface file. In addition, they must declared as a regular global variable in the “.C” (or “.h” if you prefer) files.

The two variables mainProxy and nElements are declared as readonly’s in the .ci file.

3.3 prime

Let us now write a program that really does some computations in parallel. The problem is simple: to find the number of primes in a given range say [low, high]. Let us also assume that we are supposed to do this using p chares (or virtual processors). Thus, our program should take as input low, high and p and find the number of primes between low and high. This example introduces the following Charm++ features:

1. Reductions
2. Performance visualization using Projections

We can easily parallelize the above problem by dividing the range [low, high] among the p chares and have each chare find the number of primes in its subrange. Then we can add the number of primes found by each chare to get the total number of primes. There are two variations on how the program can be written:

1. After each chare completes its calculation, it can send a message to the main chare indicating the number of primes it has found and the main chare keeps on adding these numbers. As we shall see, this approach leads to a lot of messages being exchanged between the main chare and other chares i.e. there is a lot of communication overhead associated with this method.
2. Use “reductions” to minimize the communication overhead. When we use reductions the number of messages send to the main chare equals the number of physical processors (as opposed to the number of chares, which may be vary large). Hence the communication overhead decreases.

We will try both the variations and we expect that for large number of chares, the second approach is more efficient. We will verify this using a tool called Projections that helps us analyze Charm++ programs. The primes program that does not use reductions is as follows:
```c
#include "hello.h"

/*mainchare*/
main::main(CkArgMsg* m)
{
    //Process command-line arguments
    nElements=5;
    if(m->argc >1 ) nElements=atoi(m->argv[1]);
    delete m;

    //Start the computation
    cout << "Running Hello on " << CkNumPes() << " processors with " << nElements << " elements" << endl;
    mainProxy = thishandle; // stored into a global readonly
    CProxy_Hello arr = CProxy_Hello::ckNew(nElements);
    arr[0].sayHi(17);
}

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

Hello::Hello()
{
    cout << "Hello " << thisIndex << " created" << endl;
}

void Hello::sayHi(int hiNo)
{
    CProxy_Hello arrayProxy(thisArrayID);
    cout << "Hi[" << hiNo << "] from element" << thisIndex << endl;
    if (thisIndex < nElements-1)
        //Pass the greeting on:
        arrayProxy[thisIndex+1].sayHi(hiNo+1);
    else
        //We’ve been around once-- we’re done.
        mainProxy.done();
}

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

Figure 3.5: Hello world with Chare Arrays: the .C file
mainmodule PrimesNoRed
{
    readonly CProxy_MainPrime mainProxy;
    readonly ulong low;
    readonly ulong high;
    readonly ulong numChares;

    mainchare MainPrime
    {
        entry MainPrime(CkArgMsg* m);
        entry void done(int noOfPrimes);
    }

    array [1D] WorkPrime
    {
        entry WorkPrime();
    }
};

Figure 3.6: Counting Primes: the .ci file
#include "PrimesNoRed.decl.h"

// type for readonly variables must be a single token
typedef unsigned long ulong;

class MainPrime : public Chare
{
public:
    MainPrime(CkArgMsg* m);
    void done(int noOfPrimes);

private:
    int finalCount;
    unsigned long numResponded;
    double startTime, endTime;
};

class WorkPrime : public ArrayElement1D
{
public:
    WorkPrime();
    WorkPrime(CkMigrateMessage* m) {}
};

Figure 3.7: Counting Primes: the .h file
#include "primes-nored.h"
CProxy_MainPrime mainProxy;
ulong low, high, numChares;
MainPrime::MainPrime(CkArgMsg* m)
{
    low = atol(m->argv[1]);
    high = atol(m->argv[2]);
    numChares = atol(m->argv[3]);
    mainProxy = thishandle;
    finalCount = 0;
    numResponded = 0;
    CProxy_WorkPrime work = CProxy_WorkPrime::ckNew(numChares);
    startTime = CkWallTimer();
}
// called by each chare after it has finished its processing
void MainPrime::done(int noOfPrimes)
{
    finalCount += noOfPrimes;
    // if all chares have responded, the program is done.
    if(++numResponded == numChares) {
        endTime = CkWallTimer();
        cout << "Range : [" << low << "", " << high << "]" << endl;
        cout << "Number of chares " << numChares << 
            " , number of processors " << CkNumPes() << endl;
        cout << "Number of primes = " << finalCount << endl;
        cout << "Time for processing " << (endTime - startTime) << endl;
        CkExit();
    }
}
#include "countPrimes.C"
// WorkPrime constructor: calculates number of primes in its range
WorkPrime::WorkPrime()
{
    int noOfPrimes = 0;
    ulong rangeSize = (high - low + 1) / numChares;
    // find what range am I supposed to operate on using my index
    ulong myLow = low + thisIndex * rangeSize;
    ulong myHigh = (thisIndex != numChares - 1) ? myLow + rangeSize + 1 : high;
    noOfPrimes = countPrimes(myLow, myHigh);
    mainProxy.done(noOfPrimes); // send result to main chare
}
#include "PrimesNoRed.def.h"

Figure 3.8: Counting Primes: the .C file
mainmodule Primes
{
    readonly CProxy_MainPrime mainProxy;
    readonly ulong low;
    readonly ulong high;
    readonly ulong numChares;

    mainchare MainPrime
    {
        entry MainPrime(CkArgMsg* m);
        entry void done(CkReductionMsg* result);
    };

    array [1D] WorkPrime
    {
        entry WorkPrime();
    };
};

Figure 3.9: Counting Primes: the .ci file
```cpp
#include "Primes.decl.h"

typedef unsigned long ulong;

class MainPrime : public Chare
{
public:
    MainPrime(CkArgMsg* m);
    void done(CkReductionMsg* result);
private:
    double startTime;
    double endTime;
};

class WorkPrime : public ArrayElement1D
{
public:
    WorkPrime();
    WorkPrime(CkMigrateMessage* m) {}
};
```

Figure 3.10: Counting Primes: the .h file
```cpp
#include "primes.h"
CProxy_MainPrime mainProxy;
ulong low, high, numChares;
MainPrime::MainPrime(CkArgMsg* m)
{
    low = atol(m->argv[1]);
    high = atol(m->argv[2]);
    numChares = atol(m->argv[3]);
    mainProxy = thishandle;
    CProxy_WorkPrime work = CProxy_WorkPrime::ckNew(numChares);
    // set the callback for this array
    CkCallback* cb = new CkCallback(CkIndex_MainPrime::done(NULL), thishandle);
    work.ckSetReductionClient(cb);
    startTime = CkWallTimer();
}
// reduction client -- called when the reduction is complete.
// The result of the reduction is in the 'reduction message' passed to the entry method
void MainPrime::done(CkReductionMsg* result)
{
    endTime = CkWallTimer();
    int* num_primes = (int*)result->getData();
    ckout << "Range : [" << low << "," << high << "]" << endl;
    ckout << "Number of chars " << numChares << ", number of processors " << CkNumPes() << endl;
    ckout << "Number of primes = " << *num_primes << endl;
    ckout << "Time for processing " << (endTime - startTime) << endl;
    delete result;
    CkExit();
}
#include "ountPrimes.C"
WorkPrime::WorkPrime()
{
    int noOfPrimes = 0;
    ulong rangeSize = (high - low + 1) / numChares;
    ulong myLow = low + thisIndex* rangeSize;
    ulong myHigh = (thisIndex != numChares - 1) ? myLow + rangeSize - 1 : high;
    noOfPrimes = countPrimes(myLow, myHigh);
    contribute(sizeof(int), &noOfPrimes, CkReduction::sum_int);
}
#include "Primes.def.h"
```

Figure 3.11: Counting Primes: the .C file
We will keep the range and the number of chares (p) as global variables so that all chares can access them. The mainchare has (apart from the constructor) an entry method

\textbf{done()}. This entry method will be called by other chares to indicate that it has finished processing. The main chare maintains the count of primes that the other chares have reported to it and also the number of chares that have finished processing. The program starts with the main chare creating an array of \texttt{p WorkPrimes}. Using its index, each WorkPrime finds the range its supposed to operate on and finds the number of primes in that range. It then calls the

\textbf{done()} entry method of main chare to report the number of primes and to indicate that it has done processing. In \textbf{done()}, main chare updated the total number of primes and checks if all the chares have responded. If yes, it prints the results and exits.

### 3.4 array ring

Let there be a ring of elements. We wish to send messages around this ring according to the following scheme.

\begin{verbatim}
for every element in the ring
hop-count <- m
for i 1 to k
send message to element hop-count distance away from you
hop-count <- hop-count + m
\end{verbatim}

Let’s make each of these elements a chare. However charm doesn’t specify a ring of chares. So we use our good old trusted array to simulate a ring. We build an array of chares and allow each of them to send a message to the chare it wants to in that iteration. However this requires that each element have a handle for the array of chares. This is done by having a readonly variable \texttt{elemProxy} which stores the proxy for the array of chares when the array is created. Each chare accesses this readonly variable and calculates the index of the destination chare and calls its entry method like this

\texttt{elemProxy[(thisIndex+hop-count)%numchares].recvMessage();}
Let’s make each of these elements a chare. However charm doesn’t specify a ring of chares. So we use our good old trusted array to simulate a ring. We build an array of chares and allow each of them to send a message to the chare it wants to in that iteration. However this requires that each element have a handle for the array of chares. This is done by having a readonly variable elemProxy which stores the proxy for the array of chares when the array is created. Each chare accesses this readonly variable and calculates the index of the destination chare and calls its entry method like this

\[ \text{elemProxy[(thisIndex+hop-count)\%numchares].recvMessage();} \]

### 3.5 Parallel Prefix

### 3.6 Jacobi

We have been introduced to different features of charm++ through the previous examples. We now present a problem which will make use of most of the features learnt till now and then build upon it to introduce interesting features like migration and load balancing. The problem on hand is implementing the famous Jacobi iteration to run on parallel machines.

First let’s talk about the basic Jacobi problem. There is a 2 dimensional grid, one corner of which is maintained at a constant temperature, say 1 and another at a different constant temperature, say 0. The aim is to find out what would the temperature be at the intermediate points in the grid in the steady state. Let’s call the points on the grid which are maintained at a constant temperature are called boundary points.

The algorithm used looks like this

```plaintext
do{
  for every point in the grid
    if(point is not a boundary point)
      newtemp <- average of temperature of itself
                  and neighbours on the left, right, above and below
      error<- newtemp - temp
    for every point in the grid
      temp <- newtemp
      calculate maximum error among the points on the grid
```
If a point doesn’t have all the neighbours, only those neighbours present are considered for the average.

The problem is to implement a parallel version of this algorithm.

Solving the Jacobi problem by breaking the grid into equal-sized pieces doesn’t involve many new features. It uses the old techniques of reduction, broadcast and normal entry methods. However, arrays of doubles are passed as arguments of entry methods along with the size of the arrays. The `RecieveGhost` method in the code does that. This algorithm is not very difficult to parallelize because the data dependency is not too complicated. We split the main grid into a collection of columns. Each collection of columns is placed on an element of a chare array. All the columns of a chare array element, except the first and last columns, have all the data they need. Two ghost columns are introduced to store the data columns needed to complete the calculation for the first and last columns. Each chare sends its first column to the chare on its left and the last column to the chare on its right. The receiving chares store them in their corresponding ghost columns.

The exact parallel algorithm is like this

split the grid uniformly among the chares.
do{
   for each chare
      sendghost messages to your left and right neighbours

   when a chare has received all the ghost messages it needs
   enforce boundary conditions (ie fix the values for the boundary points)
   calculate newtemp and error for every point of the grid stored in the chare

   perform reduction to find maximum error among all chares
}while(maximum error < threshold value)
3.7 Migration

3.8 Load imbalance

3.9 Load balancing

3.10 Structured Dagger