Chapter 1

The Charm++ Programming Model

Laxmikant V. Kale
Department of Computer Science, University of Illinois at Urbana-Champaign

Gengbin Zheng
National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

1.1 Design Philosophy .............................................................. 2
1.2 Object-based Programming Model .................................... 3
1.3 Capabilities of the Adaptive Runtime System .................... 7
1.4 Extensions to the Basic Model .......................................... 10
1.5 Charm++ Ecosystem ......................................................... 13
1.6 Other Languages in the Charm++ Family ........................... 14
1.7 Historical Notes .............................................................. 15
1.8 Conclusion .................................................................. 16

CHARM++ [131] is a C++ based parallel programming system developed at the University of Illinois. It has been designed and refined in the context of collaborative development of multiple science and engineering applications, as the later chapters in this book illustrate. The signature strength of CHARM++ is its adaptive runtime system, which allows programmers to deal with increasingly complex supercomputers and sophisticated algorithms with dynamic and evolving behavior. Its basic innovation is the idea of over-decomposition (explained further in Section 1.2): the programmer decomposes the computation into objects rather than processors, and leaves the decision about which object lives on which processor to the runtime system. Specifically, some of the benefits of CHARM++ to the programmer include:

- Processor-independent programming: The programmer decomposes the computation into logical units that are natural to the application, uncluttered by the notion of what data is found on which processor, and which computations happen on which processor.

- Asynchronous programming model with message-driven execution: Communication is expressed in a highly asynchronous manner, without opportunities for the program to block the processor awaiting a remote event. This model is supported by message-driven execution, where the processor-level scheduler selects only those computations for which all data dependencies have been satisfied.
2 Parallel Science and Engineering Applications: The Charm++ Approach

- **Automatic communication/computation overlap:** Without any explicit programming effort, the Charm++ runtime ensures that processors are not held up for communication, and that the communication is spread more uniformly over time. This leads to better utilization of the communication network, and to programs that are highly tolerant of communication latencies.

- **Load Balancing:** The Charm++ runtime automatically balances load, even for applications where the load evolves dynamically as application progresses. It can handle both machine-induced and application-induced load imbalances.

- **Resilience:** Charm++ applications can be automatically checkpointed to disk, and restarted on a different number of processors, within memory limits. Further, Charm++ applications can be made automatically tolerant of node failures, automatically restarting based on in-memory checkpoints when the system detects a failure, on machines where the schedulers will not kill a job if one node dies.

The purpose of this chapter is to introduce the basic concepts in Charm++, and describe its capabilities and benefits for developing complex parallel applications using it. The next chapter illustrates the process of designing applications in Charm++ with choices and design decisions one must make along the way. A more thorough tutorial on how to design Charm++ applications can be found elsewhere. For example, for online resources, see http://charm.cs.illinois.edu.

1.1 Design Philosophy

To appreciate the features of Charm++, it is necessary to understand the main design principles that were used as guidelines when developing it.

The first of these principles has to do with the question: what aspects of the parallel programming task should the “system” automate. The design of Charm++ is guided by the idea of seeking an optimal division of labor between the programmer and the system, i.e, we should let the programmers do what they can do best, and automate those aspects that are tedious for the programmer but relatively easy (or at least feasible) for a system to automate. Some parallel programming systems are designed with the view that the system should simply provide minimal mechanisms, such as basic communication primitives, and get out of the way. This has the advantage that the application developers are least constrained. An alternative is presented by the ideal of a perfect parallelizing compiler: the programmers write (or better still, just brings their own dusty deck) sequential code, and the system auto-magically
The Charm++ Programming Model

parallelizes it effectively for the target machine. The former approach is inade-quate because it does not raise the level of abstractions, while the latter has been seen to be unrealizable, despite valiant efforts. *Seeking an optimal division of labor* between the application programmer and the system has led to foundational design features in Charm++.

The second design principle is to develop features only in an application-driven manner. This is to counter a common and natural tendency among computer scientists toward a platonic approach to design, which one could call *design in a vacuum*: features are developed because they appear beautiful to their developers, without serious consideration of their relevance to a broad set of applications. To avoid this, Charm++ evolved in the context of development of parallel science and engineering applications, and abstractions or features were added to it when the application use cases suggested them [118].

1.2 Object-based Programming Model

It is important to note that although Charm++ is certainly a novel and distinct parallel programming model, different than prevailing models such as MPI, it is not a different “language” – code is still written in C++\(^1\). Programmers are only required to provide declarations of the methods that are meant to be invoked remotely, so that the system can generate code to pack and unpack their parameters automatically. Beyond that, one uses the C++ API provided to make calls into the Charm++ runtime system.

The basic innovation in Charm++ is that the computation is broken down by the programmer into a large number of objects, independent of the number of processors. These objects interact with each other through asynchronous method invocations (defined below). In such interactions and communications, it is the objects that are named explicitly and not the processors; the program is mostly free of the concept of a processor. This empowers the adaptive runtime system at the heart of Charm++ to place objects on processors as it pleases, and to change the placement during the execution of the program. This *separation of concerns between the application logic and resource management* is at the heart of many benefits that this programming model confers on application developers.

These “migratable” objects, which are the units of decomposition in the parallel program, are called *chares*\(^2\) in Charm++. Of course, a Charm++

\(^1\)Although there exist bindings for C and Fortran, we will focus on the C++ bindings in this chapter. Most of the applications in the book are written in C++. It is also, of course, possible to write most of the application in C or Fortran by using Charm++ to express all the parallel aspects, and calling sequential C and Fortran functions for the application specific code.

\(^2\)The ‘a’ in chare (\texttt{char}) is pronounced like the ‘a’ in father and the ‘e’ is silent.
program may also include regular C++ objects — but the runtime system does not need to pay attention to them. Each such sequential regular C++ object is “owned” by a single chare (Figure 1.1(a)). So, they migrate with the chare if the runtime system decides to migrate the chare to another processor. The programmer’s view of the overall computation is that of many such chares interacting with each other, as shown in Figure 1.1(b).

Let us examine a chare in isolation first, as shown in Figure 1.1(a): it is a C++ object comprising data elements and private or public methods. Its public methods are remotely invocable, and so are called “entry” methods. It is the existence of these entry methods that distinguishes a chare class from a plain C++ class. Chares can directly access their own data members, and cannot usually access data members of other chares. In that sense, a chare can be thought of as a processor as well. Since, typically, multiple chares live on a real processor, we can call them “virtual” processors. Consequently, we have called our approach the “processor virtualization” approach [127]; however, it is important to note that it is significantly different than (but related to) the relatively recent idea of OS virtualization made popular for the “cloud” infrastructure by VMWare and Xen systems.

**Asynchronous method invocation:** A running object may execute code that tells the runtime system to invoke an entry method on a (potentially) remote chare object with given parameters. The programmer understands that such method invocation is asynchronous: all that happens at the point where the call is made is that the parameters are packaged into a message, and the message is sent towards the chare object in question. It will execute at some undetermined point in future. No return values are expected from an asynchronous method invocation. If needed, the called chare will send a method invocation to caller at some time in the future.

The **execution model**, from the point of view of the programmer, is as
follows: an ongoing computation consists of a collection of chare objects and a collection of entry method invocations that have been directed at these objects. The computation begins with construction of a designated main chare. The user code in the constructor of the main chare may initialize the Read-only variables. These should not be changed by user code afterwards. The runtime system makes a copy of such variables available on each processor. The constructor of the main chare typically contains user code that create chares and collections of chares (see below), and thus seeds the overall computation.

On each processor, a message driven scheduler (Figure 1.2) in the runtime system selects one of the entry method invocations targeted at some object residing on its processor, unpacks the parameters for the invocation, and executes the entry method on the identified object with the given parameters. In the baseline model, it lets the method invocation continue to completion (see Section 1.4 for exceptions), at which point control returns back to the runtime scheduler. Since the asynchronous method invocations can be thought of as messages, this aspect of the execution model is called message-driven execution.

![Message-driven scheduler](image)

**FIGURE 1.2: Message-driven scheduler**

One of the major benefits of message-driven execution is an automatic and adaptive overlap between communication and computation. There is no call in Charm++ that will block the processor waiting for some remote data. Instead, control passes to some chare that already has some data waiting for it, sent via the method invocation from a local or remote object. The time an object is waiting for some communication from its remote correspondent is thus naturally overlapped with computation for some other object that is ready to execute.

A chare normally just sits passively. Whenever a method invocation (typically initiated asynchronously by some other chare) arrives at a chare, it executes the method with the parameters sent in the invocation. This may result in creation of some new asynchronous method invocations for other chares (or even itself) that are handed over to the runtime system to deliver. It changes its own state (i.e. values of its data member variables) as a result.
of the invocation. It then goes back to the passive state waiting for another
method invocation\(^3\).

The code inside each entry method can carry out any computation it wishes
using data completely local to its object. In addition, it can use data that has
been declared as read-only.

From the application’s point of view, a chare could be a piece of the do-
main (in “domain decomposition” methods used commonly in parallel CSE
applications). It may also be a data structure, or a chunk of a matrix, or a
work unit devoid of any persistent state. The programmer is responsible for
deciding how big the chare should be, viz. the *grainsize* of the chare. More on
that in the next chapter.

One can create a singleton chare instance dynamically, and the system will
decide on which processor to anchor it. All that happens at the call is that a
seed for the new chare is created, which captures the constructor arguments
for it; a seed-balancer dynamically moves the seeds among processors for load
balancing, until it is scheduled for execution on some processor by executing
its constructor, at which point we can assume that the chare has *taken root*
there. Chares can obtain their own global IDs (called proxies in Charm++),
and methods can be invoked asynchronously using these proxies. Parallel pro-
gramming based on such dynamic creation of individual chares is useful in a
variety of situations, including combinatorial search.

For applications in science and engineering, we need a further abstraction:
multiple chares may be organized into a collection, and each chare belonging
to a collection can be named (and accessed) by an *index*. For example, one
may have a one-dimensional array of chares. One can then broadcast method
invocations to all the elements of a collection, or to a single named one. These
collections are called *chare arrays*. However, they are not limited to be simple
arrays. The index structures may define collections that are multi-dimensional
sparse structures (e.g. a 6-dimensional array, with only a small subset of pos-
sible indices being instantiated as chares). They can also be indexed by other
arbitrary indices, such as strings or bit vectors, but such usage is not common
in current CSE (Computational Science and Engineering) applications.

A single program may contain multiple chare arrays. These may arise from
multiple application modules, or a single module whose algorithm is more
naturally expressed in terms of multiple chare arrays. To communicate with
chares belonging to a chare array, one must get hold of a “proxy” — an object
that stands for (or refers to) the whole collection. A proxy is returned when
a chare array is first created. So, the code `A[1].foo(x,y)`; specifies asyn-
chronously sending a method invocation for the method `foo` with parameters
`x,y` to the i’th object of a 1-dimensional array referenced via its proxy `A`. The

\(^3\)The model up to this point is similar to the “actor” model developed by Hewitt, Agha,
Yonezawa, and others, with the possible exception of the idea of an explicit “mailbox” that
an actor has access to. More important points of departure come in the features described
after this point, and in the reliance of Charm++ on its extensive adaptive runtime system.
call immediately returns, while the method invocation is packaged and sent to the processor where the $i^{th}$ element resides.

Chare arrays support reductions and broadcasts over all its elements, but, unlike in MPI-1 or MPI-2, these are both asynchronous non-blocking operations. (MPI-3 standard has now adopted non-blocking collectives). A broadcast can be initiated by any element or even from other chares not belonging to the target chare array. In our example above, $A\.foo(z,t)$ will result in asynchronous invocations of the $foo$ method of all the member chares of $A$ — a broadcast. The system ensures that all the chares belonging to a chare array receive the successive broadcasts in the same sequence. Reductions are carried out via non-blocking “contribute” calls that allow other computations to proceed while the result of the reduction (such as a global sum) is delivered to its intended target, via an entry method invocation or via a general-purpose callback abstraction. In particular, the members of the chare array over which the reduction is being carried out are free to execute other entry methods while the reduction is in progress.

The chares belonging to a chare array are assigned to processors by the runtime system (RTS) as shown in Figure 1.3; the RTS may change this assignment at runtime as needed. A scalable location manager [153] keeps track of which chares are on which processor, resulting in messages being delivered quickly and with low overhead to the right processor.

FIGURE 1.3: System view of chares
1.3 Capabilities of the Adaptive Runtime System

The heart of the Charm++ system, and its signature strength, is its adaptive runtime system. The primary responsibilities of the runtime system are:

1. To decide which objects reside on which processor, when they are created
2. To schedule (i.e. sequence) the execution of all pending entry method invocations on the targeted chare objects on each processor,
3. To keep track of current location of each chare, in spite of chare migrations, in a scalable and low-overhead manner,
4. To mediate communication between chares by delivering entry method invocations to the correct target object on the processor where it resides.
   And, finally
5. To migrate chares across processors, if needed, in response to runtime conditions.

The Charm++ programming model provides much flexibility to the runtime system, in terms of placement of chares on processors, sequencing of method invocations, and mediating and intercepting communication between chares. The Charm++ adaptive runtime system (RTS), thus empowered, leverages this flexibility to optimize performance as the program executes. Here, we will briefly discuss its capabilities in balancing load dynamically, tolerating faults, optimizing communications, and managing power.

**Dynamic Load Balancing:** Charm++ supports a large suite of load balancing strategies. Some of these strategies use measurements of computational loads and communication graph between chares, which the RTS can readily obtain because of its role in scheduling chares and mediating their communication. With Charm++, load balancing can be thought of as a two-phase process: the programmer decomposes the work (and data) into chares. This division does not have to be perfect: i.e. significant variation in the work/size of chares is permissible, since there are typically tens of chares on each processor core. At runtime, the RTS assigns and reassigns chares to individual processors, to attain such goals as better load balancing, and/or minimization of communication volume. As the number of chares is much smaller than the number of underlying data-structures elements (e.g. grid points, or mesh elements), the load balancing decisions are much faster than, say, applying a graph partitioner such as ParMETIS to the entire underlying structure. Occasionally, chares may have to be split or merged to keep their size within a desired range: Charm++ supports dynamic creation and deletion of chare array elements if needed. But this is not needed for most applications, and when
The Charm++ Programming Model

its needed, it is still simpler than a complete repartitioning of the application data structures.

The suite of strategies provided with Charm++ includes some that ignore communication volume and some that consider it. It also includes some strategies that refine load balance, by moving a relatively small number of objects from overloaded processors, and other schemes that comprehensively repartition the object graph. For large machines, it includes strategies that optimize placement with respect to the interconnect topology, and hierarchical strategies that significantly reduce decision time. Further, one can write application-specific strategies (or new, general-purpose ones) using a relatively simple plug-in architecture. Also, a meta-balancer that examines application characteristics, to choose the appropriate strategy, and decide when to apply it, has been developed recently.

Automatic Checkpointing: Parallel application developers often need to write code for periodically checkpointing the state of their application to the disk. Simulation runs are often long, and need to be broken down into segments that will fit within system-allowed durations; also, hardware failures may cut short an ongoing simulation. Checkpoints allow one to handle such situations without losing much computation. Since Charm++ already has the capability of migrating objects to other processors (with users providing information to optimize the amount of data saved, if needed), the RTS can leverage this capability to “migrate” copies of objects to the file system, along with the state of the runtime system itself. This reduces the burden on the programmer as they do not need to write additional checkpointing code. Furthermore, when restarting, they can use a different number of processors than what was used for the original simulation, e.g., a job that was running on 10,000 cores can be restarted on 9,000 cores! This works automatically for baseline Charm++ programs, and requires little extra programming for programs with user-defined groups and node-groups (Section 1.4).

Fault tolerance: One can also make a Charm++ application continue to run in spite of individual nodes crashing in the middle of the execution! Charm++ offers multiple alternative schemes for this purpose. The most mature, and probably most useful for applications today, is the double checkpointing scheme, which stores a checkpoint of each object locally and on a buddy node. An automatic failure-detection component checks the “heartbeat” of each node in a scalable and low-overhead manner. When a node fails, the system effects a recovery by automatically and quickly restoring the state of the last checkpoint. How quickly? We have measured restarts in hundreds of milliseconds for a molecular dynamics benchmark on over 64k cores [260]! Even on applications with very large checkpoints, it usually takes no more than a few seconds. One can use spare processors or make do with remaining processors on failure. For large runs, running with a few spares is inexpensive and simplifies the load balancing the system must do after restart.

A more advanced scheme based on message-logging with parallel restart
10 Parallel Science and Engineering Applications: The Charm++ Approach

has also been developed [37, 36, 175]. With the double-checkpoint scheme (as with any other checkpoint-restart scheme), when a node fails, all the nodes must be rolled back to their checkpoints, wasting energy and wasting a lot of computation. With our message-logging schemes, when a node fails, only its objects restart from the checkpoint, while the others wait. The restarting objects can recover in parallel on other processors, thus speeding recovery. It does require storing of messages at the senders, which can add to memory overhead. Many strategies aimed at reducing this overhead have been developed [176]. This scheme is expected to be more important beyond Petascale, when node failures are likely to be frequent.

**Power Management:** Power, energy and temperature constraints are becoming increasingly important in parallel computing. Charm++, with its introspective runtime system can help by monitoring core temperatures and power draw, and automatically changing frequencies and voltages. It can rely on its rate-aware load balancers (i.e. strategies that take into account the different speeds of different processors) to optimize either execution time or energy, while satisfying temperature and total-power constraints. As an illustration [219, 220], we were able to reduce cooling energy in a machine room by increasing the A/C thermostat setting; of course, that may lead to some chips overheating. However, the Charm++ runtime system monitored chip temperatures periodically, and lowered the frequencies of chips that were getting too hot, while increasing them if they were cold enough. This creates a load imbalance which would slow the whole application down, as the rest of the processors wait for data from the processor whose frequency was lowered. However, the runtime is able to rebalance load by migrating objects after such frequency changes. These power-related features are available only in experimental versions of Charm++ at the time of this writing, but are expected to be more broadly available in near future.

**Communication optimizations:** The Charm++ runtime system is continually observing the communication patterns of the application, since it is delivering messages to chares. It can replace communication mechanisms based on the observed patterns. For example, algorithms for collective communication can be changed at runtime, between iterations of an application, based on the size of messages, number of nodes, and machine parameters [144].

1.4 Extensions to the Basic Model

In section 1.2 we described the basic Charm++ programming model, consisting of chares, indexed collections (arrays) of chares, asynchronous method invocations, broadcasts and reductions. This baseline description is very useful
for developing an intuition about the programming model, and its underlying operational semantics (or execution model). A few important extensions to the base model, which enrich the programming model without changing its essential character, are noted below. These “extensions” are as mature and old as Charm++ itself, and are in common use in applications today.

**Supporting “blocking” calls:** Charm++ supports two additional kinds of methods, specifically tagged as such, that do allow “blocking” calls. They do not block the processor; only the affected entry method is paused, and control is returned to the scheduler. These are called **Structured Dagger** methods and **threaded** methods, as explained below.

A **Structured Dagger** (also abbreviated sdag) entry method allows users to define a DAG (directed acyclic graph) between computations within a chare, and asynchronous method invocations expected by the chare. This typically allows one to express the life cycle of a chare object more clearly than a baseline program would. An important statement in the structured-dagger notation is the so-called **when** statement, which specifies (1) that the object is ready to process a particular entry method invocation, and (2) what computation to execute when this method invocation arrives.

Just to give a flavor of how sdag code looks like, we present a snippet of code below. This comes from a molecular dynamics example, discussed briefly in the next chapter. But that is not important here; we are just illustrating the structure of sdag code. The “run” method of this chare includes a time step loop. In each time step \( t \), the run method waits for two invocations of coordinates method, and when both are available executes some sequential object methods atomically. The sequential code calculates forces on each set of atoms \( C_1 \) and \( C_2 \) due to the other set of atoms, and sends the resultant forces back. Since this is not usual C++ code, sdag entries are specified in a separate file, which is translated into C++ code. One can thus think of sdag code as a script for describing data-dependent behavior of a chare. Typically, it describes the entire life cycle of a chare, as signified by the name, “run” method, in this particular case.

```cpp
entry void run() {
  for (t=0; t<steps; t++) {
    when coordinates(vector <Atom> C1),
    coordinates(vector <Atom> C2)
    serial {
      calculateInteractions(C1, C2);
      sendForcesBack();
    }
  }
}
```

When a **threaded method** is invoked the runtime system creates a lightweight user-level thread and starts a method invocation inside this thread. A threaded entry method can block waiting for a future [99], or for a return value from a **synchronous** method invocation. Correspondingly, the system al-
allows users to define entry methods that return a value, as well as a simple future abstraction. One can create a future, set value to it, or access value from it (which is a blocking call). If a thread tries to access the value of a future, and the value is not set yet, the thread is blocked, and control is transferred to the Charm++ scheduler. Later, when the value is set, the thread is added back to the scheduler’s queue, so it can be resumed in its turn.

**Array Sections:** A subset of chares belonging to a chare array can be organized into a section, somewhat like an MPI sub-communicator. One can invoke broadcasts and reductions over sections as well. The system organizes efficient spanning trees over the subset of processors that house elements belonging to a section. It ensures that the broadcasts and reductions are carried out correctly even when element chares migrate to other processors, and reorganizes the spanning trees periodically, typically after a load balancing phase.

**Processor-awareness:** Another extension has to do with awareness of processors by the programmer. In the model described so far, there is no need for the programmer to know anything about the processors, including which processor is the current location of a particular object. However, there are some situations in which an “escape valve” into processor-aware programming is needed. This is especially true for libraries, or performance oriented optimizations. For example, many objects on the same processor may request the same remote data; it makes sense in this situation to use a shared processor-level software cache; Requests for remote data can go via this cache object, and if requested data was already obtained due to another chare’s request, unnecessary remote communication is avoided. For such purposes, Charm++ provides a construct called chare-group. Just like an array of chare objects, a chare-group is a collection of chares. However, (1) there is exactly one member mapped to each processor, and (2) unlike regular chares, chare group members are allowed to provide public methods that are invoked directly, without needing the packaging and scheduling of method invocations. Also, given the group ID, the system provides a function that returns a regular C++ pointer to the local (branch) chare of the group. With these two features, chares can communicate using low-overhead function calls with the member (“branch”) of a group on their processor. Note that such group objects also allow additional data sharing mechanisms [228] beyond the read-only variables mentioned earlier.

So far, we intentionally left the notion of what we mean by a “processor” only loosely defined. In Charm++, for processor-aware programming, there is a notion of PE (processing element). A PE corresponds to a single scheduler instance; a Charm++ application may associate a PE with a hardware thread, a core, or a whole or a part of a multicore node, based on command-line options. If a PE includes multiple hardware resources (say cores), the parallelism within a PE is managed by the user orthogonally, by using pthreads, openMP, or Charm++’s own task library (called CkLoop). Associating a PE with a hardware thread is a common practice in current Charm++ applica-
tions, and it obviates the need to deal with an additional level of parallelism, so we will assume this in our description.

Of course, the group construct and other such low-level features should be used sparingly. As a design guideline, one should strive to avoid using processor-aware programming as much as possible, and push it into low-level libraries when needed. The example in the above paragraph, involving multiple requests for remote data, is a common enough feature that a new library, CkCache, has been developed as a common library for use by multiple applications. The system libraries for implementing asynchronous reductions are another example. Although one could implement a spanning tree over all the chares of a chare-array, it is much more efficient to do a processor (and node) based spanning tree, collecting inputs from all the local chares first.

Since objects may be migrated by the runtime system to other processors, Charm++ also supports a special callback method that gets called after the object has been re-incarnated on another processor; this can be used to update any processor-specific information, such as pointers to local branches of groups, stored by the objects.

1.5 Charm++ Ecosystem

Charm++ is a mature and stable parallel programming system. Thanks to the popularity of applications such as NAMD, it is used by tens of thousands of users worldwide (The biomolecular simulation code NAMD, described in Chapter 4, has 45,000 registered users, as of December 2012). Charm++ is available on most national supercomputer installations in the US. Charm++ runs on almost all the parallel computer types that are widely known, including Cray machines, IBM Blue Gene series machines, Linux Clusters, Windows clusters etc. It supports multiple network types including proprietary networks on supercomputers, as well as commodity networks including Ethernet and Infiniband. Charm++ is regression-tested via a nightly build system on dozens of combinations of compilers, operating systems, processor families and interconnection networks.

The maturity of Charm++ is also reflected in the ecosystem of program development and analysis tools available for it. Projections is a sophisticated performance visualization and analysis tool. CharmDebug is a more recent and highly sophisticated debugging tool. In addition, the LiveViz library can be used to collect application or performance data during application run and visualize it as the program is running. The CCS (Converse Client-Server) library that underlies LiveViz also allows one to develop interactive parallel applications, whereby queries or messages can be injected into a running computation, either to examine specific attributes of a running simulation, or to effect changes in the execution of the application.
There are several online resources for learning Charm++ and working with it. The software, manuals, tutorials and presentations, are available at http://charm.cs.illinois.edu. An active mailing list (charm@cs.illinois.edu) is used for reporting bugs and discussing programming issues and upcoming features. There is an annual workshop on Charm++ and its applications in Urbana Illinois; the presentations from the workshop (starting in the year 2002), most including the video of the presentations, are also available online.

1.6 Other Languages in the Charm++ Family

Charm++ is just one instance of a broader programming paradigm based on message-driven execution, migratable work and data-units, and an introspective and adaptive runtime system. Although Charm++ is the earliest member of this family of programming languages there are a few others that we have developed that deserve a mention here. All of these are built on top of Charm++, as Charm++ turns out to be an excellent backend for developing new abstractions within this broad paradigm.

XMAPP is the name we have chosen for the abstract programming model that underlies Charm++ as well as all the other languages described below. XMAPP is characterized by a few defining attributes:

- Over-decomposition: the interacting entities, be they units of work or units of data (or a mix of the two, as in Charm++), into which the computation is decomposed by the programmer in such models are independent of the number of processors, and typically their number is much larger than the number of processors.

- Processor-independence: the interaction/communication between entities is in terms of names of those entities and not in terms of processors.

- Migratability: these entities can be migrated across processors during program execution, either by the runtime system, or the application itself, or both.

- Asynchrony: collectives and other communication-related operations are designed so that their implementations do not block the processor.

- Adaptivity: the runtime system takes responsibility of balancing load by leveraging its ability to migrate objects.

Adaptive MPI (AMPI) is an implementation of the MPI standard on top of Charm++. In MPI, the computation is expressed as a collection of processes that send and receive messages among themselves. With AMPI, each
The Charm++ Programming Model

MPI process is implemented as a user level thread. These threads are embedded inside Charm++ objects, and are designed to be migratable across processors with their own stack. As with Charm++, multiple “processes” (i.e. MPI ranks) are typically mapped to a single core. Standard MPI calls, such as those for receiving messages, provide natural points to allow context switching among threads within a core, thus avoiding complexities of preemptive context switching. AMPI programs have shown to have comparable performance (somewhat slower for fine-grained messaging, but comparable for most applications) as the corresponding MPI program, even when no AMPI-specific features are being used. Those features, such as over-decomposition (and adaptive overlap of communication with computation), asynchronous collectives, load balancing, and fault tolerance, provide the motivation for using AMPI instead of plain MPI implementations.

MSA (Multiphase Shared Arrays) [51, 179] is a mini-language on top of Charm++ that supports the notion of disciplined shared memory programming. It is a partitioned global address space (PGAS) language. The program here consists of a set of migratable threads and a set of data arrays. The data arrays are partitioned into user-defined “pages”, which again are migratable data units implemented as chares. The main point of departure for the language is the notion of access modes. Each array may be in one of the few possible modes, such as “read-only” or “accumulate”. All the threads must collectively synchronize to switch the mode of an array. This model is shown to avoid all data races, and yet captures a very large fraction of use cases where one would find shared global data useful.

Charisma [108] is a language that allows elegant expression of applications or modules that exhibit a static data-flow pattern. The computation is divided into chares. If the chares exchange the same set of messages (with different lengths and contents, to be sure) in every iteration, one can express the lifecycle of entire collections of chares in a simple script-like notation, where entry-methods are seen to publish and subscribe to tagged data.

Charj [179] is a compiler supported language that provides the same abstractions as Charm++ combined with MSA. With compiler supported static analysis, Charj provides a more convenient and elegant syntax, automatic generation of serialization code, and several other optimizations based on static analysis. Charj is an experimental or research language at the current time.

1.7 Historical Notes

The precursors to Charm++ (the “Chare Kernel”) developed by us were aimed at combinatorial search applications, and at supporting parallel func-
tional and logical programming. However, once we turned our attention to science and engineering applications, we decided to mold our abstractions based on the needs of full-fledged and diverse applications. The first two applications examined were fluid dynamics [95] and biomolecular simulations [116]. These two (along with many small examples, and parallel algorithms such as the Fast multipole algorithm, histogram-based sorting [120, 142]) adequately demonstrated to us that our approach was avoiding the trap of being too specialized. This was especially true because we considered full-fledged applications, in addition to kernels or isolated algorithms. We thought that only by immersing ourselves in the nitty-gritty of developing a full-fledged application, would we be able to weigh the importance and relevance of alternative abstractions, and capabilities.

This position and approach towards development of abstractions were explicitly written down in a position paper around 1994 [118]. The biomolecular simulation program NAMD funded by NIH (and NSF, in the early days, under the “Grand Challenge Application Groups” program), provided us a good opportunity to practice and test this approach. NAMD was developed in collaboration with Klaus Schulten, a biophysicist with a computational orientation, and Bob Skeel, a numerical analyst, both Professors at University of Illinois.

1.8 Conclusion

We believe that Charm++ and the underlying XMAPP abstract programming model constitute an approach that is ready to deal with the upcoming challenges in parallel computing, arising from increasingly complex hardware, and increasingly sophisticated applications. It appears to us that the basic concepts in XMAPP are going to have to be inexorably adopted by the community, whichever language they choose to use in future. So, why not Charm++? Charm++ itself is a production-quality system that has demonstrated its capabilities in improving programmer productivity and in attaining high scalability on a wide variety of the parallel applications in science and engineering, as demonstrated by this book. Some applications have demonstrated scaling beyond half a million processor cores by now.

To simplify and ease adoption, Charm++ supports interoperability with MPI; some modules can be written in regular MPI, while others can be based on Charm++ or AMPI (or any of the other mini-languages in the Charm++ family). We invite the readers to experiment with this approach by writing modules of their applications in it, or by using an existing Charm++ library in their MPI application, or testing it by developing an isolated algorithm using it, and then possibly moving on to developing entire applications using Charm++, and reap the productivity and performance benefits.


244 Parallel Science and Engineering Applications: The Charm++ Approach


246 Parallel Science and Engineering Applications: The Charm++ Approach


[49] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. *The CHARM (5.9) programming language manual*, 2006.

[50] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. *The CONVERSE programming language manual*, 2006.


Parallel Science and Engineering Applications: The Charm++ Approach


Contagion Diffusion with EpiSimdemics


Parallel Science and Engineering Applications: The Charm++ Approach


Orion Lawlor, Hari Govind, Isaac Dooley, Michael Breitenfeld, and Laxmikant Kale. Performance degradation in the presence of subnormal floating-point values. In *Proceedings of the International Workshop*


260 Parallel Science and Engineering Applications: The Charm++ Approach


Parallel Science and Engineering Applications: The Charm++ Approach


