COMPILER SUPPORT FOR PRODUCTIVE MESSAGE-DRIVEN PARALLEL PROGRAMMING

BY

AARON KARL BECKER

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Research Assistant Professor Maria Garzaran
Professor David Padua
Professor Ponnuswamy Sadayappan, Ohio State University

ABSTRACT

Historically, the creators of parallel programming models have employed two different approaches to make their models available to developers: either by providing a library with hooks for common programming languages, by developing a new language and associated infrastructure altogether. Despite the flexibility of the language approach and the great number of parallel languages that have been created, the library approach, as exemplified by the Message Passing Interface, has dominated large-scale high performance computing.

It is our hypothesis that the combination of a rich runtime system and a relatively simple compiler infrastructure can significantly improve programmer productivity without compromising performance. In this work, we examine this hypothesis through the lens of Charj, a simple language based on the Charm++ runtime system. We consider the effect that the addition of a compiler has on user experience in terms of the ways in which features are exposed to the programmer and in opportunities for optimization, and code simplification, and the integration of multiple programming models, drawing from our experiences developing the Charm++ runtime and the Charj language. We substantiate our conclusions through the development of Charj applications that are significantly more simple than their Charm++ equivalents without sacrificing performance.

To my family.
You never asked for a dissertation, but here one is anyway.

ACKNOWLEDGMENTS

The work in this dissertation is a small part of a decades-long research agenda carried out by the members of the Parallel Programming Laboratory. As such, I owe a great debt to my friends and colleagues in the lab who have collaborated with me and provided invaluable feedback and advice over the years that I have spent there.

In particular, I would like to thank Pritish Jetley, Jonathan Lifflander, Philip Miller, and Minas Charalambides for their help during the process of designing and implementing Charj and applications based on Charj. I would also like to thank David Kunzman and Chao Mei for their support in the process of actually writing this dissertation. I am also very thankful to my dissertation committee for their valuable suggestions and comments. Finally, I owe a great debt of gratitude to my advisor, Laxmikant Kalé, who always seems to have several solutions to every problem I encounter.

I would also like to acknowledge the use of the parallel computing resource provided by the Computational Science and Engineering Program at the University of Illinois. I performed performance testing for Charj applications and their Charm++ equivalents using the Taub cluster, which is part of the CSE computing resource.

Finally, I would like to thank my family and friends, who kept me mostly sane during the process of writing and researching, put up with my complaints, and who convinced me that I would eventually succeed. I don't trust myself to list your names without forgetting anyone, but I couldn't have done it without you.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In many ways, high performance computing (HPC) remains the wild west of the programming world. While ever-growing performance and the inevitable march of Moore's Law has led to the increasing popularity of managed code, garbage collection, and dynamic typing in mainstream programming, the developers of high performance parallel applications make very few concessions to speed, and as a result they pay a high price in development and maintenance time.

Considering the intrinsic difficulties of HPC and the demands upon HPC programmers, it can be no surprise that programmer productivity in this area is notoriously poor [1–4]. Sadly, no dramatic solution to this problem has been found, and none seems likely to present itself in the near future. Indeed, even measuring exactly what one means by productivity in HPC can be a difficult problem to solve [5–7].

Under these circumstances, we must strive to relieve the programmer of as many burdens as is practically possible. The Message Passing Interface (MPI) takes the approach of giving the programmer maximal control, to the point that it has been called the assembly language of parallel computing [8]. While this approach makes it possible to write extremely successful parallel programs, it is also widely blamed in the computer science community, whether fairly or not, for creating many of the productivity problems that we aim to remedy [9]. On the other end of the spectrum, parallelizing compilers have promised to automatically extract parallelism, giving the programmer

little or no control over the parallel structure of their code. While this approach sounds appealing, in practice attaining real performance and scalability outside of narrow problem domains has not been possible without a real investment of time and effort by human programmers [10–12]. Although the intrinsic complexities of HPC software may always remain, we can at least aim to remove as much of the tiresome drudgery of programming as we can. We cannot expose the programmer to all of the overwhelming complexity of a modern HPC execution environment, nor can we hide all of the complexity behind abstractions and automation. We must rather seek a productive division of labor between the programmer and the system that provides useful abstractions without taking away the programmer's control.

This raises a natural question: how can we make it easier to write high performance parallel code? Many years of research has been dedicated to this question, and many answers have been provided, some successful and others not. Research in parallel applications has yielded a wide variety of programming models, dozens of languages, auto-parallelizing compilers, and a variety of parallel runtime systems. However, it is often difficult to see how these pieces fit together to improve the experience of actual application developers, or if in fact the pieces can be made to fit at all.

Thus far, the bulk of HPC programmers have been indifferent to the great variety of research at least partially dedicated to improving their lives. This fact argues for an approach that is more focused on the practical aspects of HPC application development and on minimizing the difficulties of adopting new tools and techniques.

It is our hypothesis that the combination of a rich runtime system and a relatively simple compiler infrastructure can significantly improve programmer productivity without compromising performance. We believe that well-known compiler techniques can be applied to carefully targeted areas to significantly simplify the development process for high performance parallel applications, and that this process need not produce less efficient code. In particular, the features exposed by a rich parallel runtime system can be made simpler, more user-friendly, and less error-prone while maintaining high performance. Rather than attempting to use the compiler to apply sophisticated optimizations or dramatic restructuring of the developer's code, we will identify areas in which we can simplify common tasks, facilitate interoperability between program modules, and support such high-level application

features as load balancing and fault tolerance through compiler support. It is our hope that by focusing on such practical considerations on a platform that is already widely used in the real world that we really can reduce the amount of blood, sweat, and tears that HPC developers must pour into their creations.

CHAPTER 2

METHODOLOGY

The primary goal of this research is to investigate the ways in which programming language and compiler support can improve programmer productivity when writing parallel HPC applications. We pursue this goal by creating a new language called Charj and an associated compiler which incorporate syntax, semantic analysis, and optimizations targeted at HPC code. We then use Charj to develop small-scale but fully functional HPC codes that are representative of a variety of common problem domains, and compare the resulting code to equivalents written using popular existing frameworks.

However, to demonstrate the usefulness of applying compiler technology to parallel-specific productivity problems, one must first decide what programming environment to target. Endless choices are possible. The type of language, the particular language syntax, the compiler framework, the optimizations to pursue–there are a huge number of variables.

Our solution space is highly constrained because of the nature of our goals. For example, if we want to create a programming environment that is broadly acceptable to current HPC programmers and that can leverage existing runtime infrastructure, it would be very problematic to create a purely functional programming language. For many of these variables, however, there is no provably right or wrong choice to be made, and so we must be guided by our notion of what will prove most expedient and practical in the demonstration of our thesis. However, even though we cannot provide logical proof that our choices are correct, we can at least provide our rationale, in the form

of guiding principles that we have used when designing the Charj language and its compiler infrastructure.

2.1 Objectives

In this chapter we discuss our goals in creating a new parallel programming environment and the ways in which our goals have informed our choices about the nature of the Charj programming language, its runtime, and its compiler infrastructure. Broadly, we aim to develop a programming environment that has four key features. First, it must have practical utility for working HPC developers. When we are faced with a choice between theoretically interesting features and practically useful features, we opt for practical utility. Second, it must effectively integrate high-level parallel features, giving the programmer simple and elegant access to complex tasks like load balancing and fault tolerance. Third, it should provide a concise and elegant syntax for expressing parallelism. Parallel operations should be smoothly integrated into the language design and not be tacked on as second-class citizens. Finally, Charj should reduce the burden on the programmer by automating tasks that are routine but effort-intensive or error-prone, especially when those tasks are related to communication.

2.1.1 Practical Utility

With Charj, we set out to create a programming language that is useful to the HPC community in practice, not only in theory. Usefulness ultimately depends on a large number of factors with little or no connection to our research agenda, such as the development of a vibrant user community and adoption by prominent users and applications, so of course our work is not and cannot be sufficient to guarantee that Charj *will* be practically useful.

Conscientious design is nevertheless necessary to allow the possibility that Charj could be broadly adopted in the HPC community. HPC programmers are known for being relatively conservative in their adoption of new technology. The Message Passing Interface (MPI), the most broadly used library for enabling parallelism in HPC applications, dates back over twenty years, and the mathematical kernels relied on by many scientific HPC applications

are still written in Fortran [13]. If Charj were to represent a complete break with existing HPC programming practice then it would have slim hopes for practical use.

Of course, in order to increase productivity in a significant way, Charj must differentiate itself from the alternatives it aims to supplant. Indeed, we must aggressively pursue opportunities to improve on the status quo. However, an appreciation for the comfort of existing HPC programmers will tend to lead us to make changes which primarily simplify or eliminate common HPC development tasks rather than making wholesale structural changes to the practice of HPC programming. Therefore, to facilitate acceptance by the existing community of HPC programmers, Charj must have familiar and easily recognizable syntax.

Syntax is a common sticking point for programmers, and minor differences in programming language syntax can lead to endless debate over aesthetics. For example, the inclusion of significant whitespace in Python has spawned reams of debate, ranting, and discussion by both Python supporters and detractors, over the years. Discussion on this topic has far outweighed discussion on more consequential matters of expressiveness and performance. This is not to say that significant whitespace is good or bad, only that this type of concern over aesthetics is important to programmers, sometimes even more important than more ostensibly substantive issues[1]. Where possible, we adopt familiar, recognizable syntax, and make as few changes as possible relative to the most widely known and used languages, which in the case of Charj means that the syntax is very similar to Java, or a subset of C++.

If HPC programmers are conservative with regard to technology choices, they are far more conservative (and understandably so) when it comes to performance. A huge amount of time and effort goes into optimizing HPC codes, and programmers are extremely reluctant to trade away any of their performance gains. This points to two key characteristics that Charj must have. First, it must produce efficient baseline code. That is, straightforward

[1]It is difficult to compare the volume of discussion on significant whitespace in Python versus the volume of discussion on more substantive Python language issues in any rigorous way. However, it is suggestive that on the c2.com wiki, a popular site for programming-related discussions, the combined size of the pages "Python Language", "Python Philosophy", and "Python Discussion" is 5764 words as of May 2012, while the combined size of "Python White Space Discussion" and the related page "Syntactically Significant Whitespace Considered Harmful" is 11072 works.

Charj code that performs basic communication must have high performance. The infrastructure must be sound. Second, Charj must accommodate programmers who wish to optimize performance-critical code by hand. It must have a reasonably transparent programming model and it should allow programmers who want to invest significant time and effort into optimization to do so effectively.

The requirement for high performance dictates that Charj must have a well-optimized messaging infrastructure. Building a high performance messaging subsystem that works across the variety of specialized networking hardware found in modern supercomputers is a very difficult undertaking in itself, and one that is largely orthogonal to the issues that we wish to address in Charj. Therefore it is much more efficient to adopt an existing messaging infrastructure for Charj.

Selecting a well-established communication framework has an additional benefit: compatibility with existing code. A large amount of time and effort has been sunk into creating highly tuned parallel code using existing frameworks, and the ability to take advantage of this code is an important factor in determining the acceptability of Charj in practical use. By sharing a common foundation with a body of existing code, Charj applications can more easily integrate with existing applications and libraries.

2.1.2 Integrating High-Level Parallel Features

HPC applications are constantly growing larger and more complex. At the same time, supercomputers are themselves growing larger and more complicated, becoming more heterogeneous and more topologically differentiated even as their increasing size drives down mean time to failure to the point where applications must have a strategy for gracefully recovering from errors. In this environment, application developers must struggle to implement features like fault tolerance and dynamic load balancing in their applications.

Although each individual application will have its own unique needs, the prerequisites for implementing such features are typically similar. They involve the need to identify key application data structures and relocate them, with sensitivity to the parallel structure of the application. By integrating these tasks into the Charj programming environment so that the compiler

has some understanding of the high-level tasks that the programmer may be attempting to perform, we believe that we can significantly improve the ease with which a programmer can produce a successful implementation.

2.1.3 Concise and Elegant Syntax

We have already claimed that Charj should have familiar syntax, because familiar syntax makes it easier to use for the existing community of HPC programmers. However, this proviso mainly applies to the syntax of serial code implemented in Charj. Nearly all HPC code is written in a language with no syntactic support for parallelism, typically C, C++, or Fortran. Although there are exceptions to this rule, such as Co-Array Fortran, they have not yet been widely adopted. In order for Charj to look and feel familiar, sequential Charj code should resemble existing sequential code to the extent possible.

However, most parallel operations in existing HPC applications are performed via library calls. There is no parallel-specific syntax for Charj to emulate. In these cases Charj should provide the simplest possible interface to the parallel functionality. Ideally all new parallel syntax should feel like part of an organic whole with the familiar serial syntax. Redundancy should be minimized.

At the same time, it is important to be able to easily distinguish serial operations from parallel operations. Especially as application and machine sizes grow, the performance implications of each parallel task can be enormous, and it is essential that the programmer can easily discern which sections of the code are purely local and which sections involve communication.

While it is important to pursue simple, elegant expressions of the programmer's intent, we must be careful not to unduly degrade performance for the sake of elegance. Often, the complexity of HPC programs are due to the need for careful performance optimizations, and while they can be simplified significantly, these simplifications come at the cost of their speed [14]. While there is often a trade-off to be made between elegance and performance, in the world of high performance computing we must err on the side of performance and carefully justify any slowdowns or inefficiencies that we introduce in the name of simplicity.

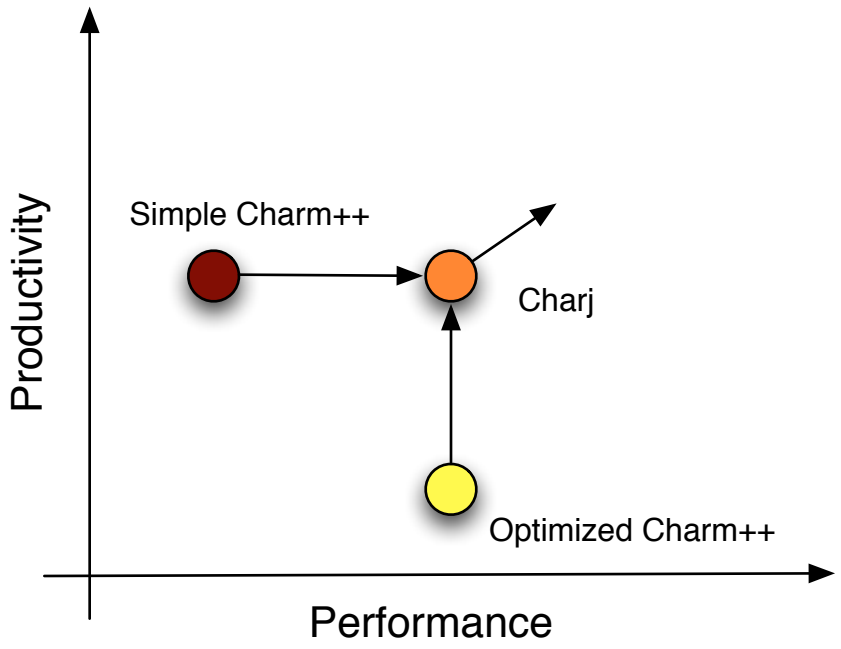


Figure 2.1: Charj aims to combine the performance associated with a well-optimized, sophisticated Charm++ application with the productivity associated with a more naïve, unoptimized approach.

2.1.4 Reducing Programmer Burden

Ultimately, each goal that we have described for Charj can be considered a part of a larger, more encompassing goal: that of reducing burden on the programmer. Programmers experience many kinds of burdens in the course of developing an application, and much of the history of the compiler could be summarized as an attempt to alleviate these burdens. Generally, we aim to reduce or eliminate programming tasks that are repetitive, mechanical, and error-prone, via syntactic analysis and code generation.

With Charj, we aim to allow programmers to achieve the performance associated with carefully written, hand-optimized Charm++ applications, but with much less time and effort. If effect, we hope to combine the productivity that programmers experience when writing relatively simple, naïveapplications while enjoying performance that would normally require a much greater investment of time, effort, and expertise, as shown in figure 2.1. Eventually we hope to use sophisticated optimizations that would be extremely cumbersome to apply by hand to extend the performance that Charj applications can realize beyond the level of even well-optimized Charm++

applications.

2.2 Libraries versus Languages

For any software environment for parallel programming, there is a question of whether it is best implemented as a language, or as a library for an existing language. There are substantial benefits to a library-based approach, especially in terms of likelihood of adoption and effort required by users. Certainly MPI, the most successful parallel programming model yet developed, makes a strong case for implementation via libraries. Its ubiquity is bolstered by the fact that programmers can use it directly in programs of a wide variety of languages. Similarly, Intel's Threaded Building Blocks (TBB) [15] and Concurrent Collections (CnC) [16] aim to attract existing C++ programmers, and their implementation as C++ libraries gives them a much lower barrier to entry than a new programming language which implemented the same parallel programming model would have.

In the case of Charj, we are building a language on top of an existing runtime library, and it is important to justify the time and effort overhead associated with that decision. This overhead manifests itself both in the development cost of the Charj language, compiler, and associated infrastructure, and in the costs incurred by programmers who must learn the Charj language in order to make use of it. In order to justify this overhead, we must provide convincing features that could not be delivered if we simply spent more time improving upon the Charm++ runtime libraries.

Charj aims to provide value that cannot be delivered via library in three key areas: syntax, safety, and speed. The first area, syntax, is an area where a new language has a decided advantage over a library-based approach, especially for libraries in commonly used HPC languages such as C, C++, and Fortran. We can provide syntax that maps directly onto key programming model concepts, and are not limited by syntax developed with only sequential execution in mind. This allows us to make parallel operations visually distinct from serial operations, where in a library-based approach, the calling of a function which invokes parallel behavior will look no different syntactically than the calling of a function which is purely sequential. Some languages are more conducive to the introduction of new syntax specific to some particular

task. For example, Ruby libraries and applications can introduce new syntax embedded into the core language transparently, and thereby gain some of the advantages that we experience in using a new language [17]. However, since Ruby is unsuitable for HPC programming, and commonly-used HPC programming languages do not easily allow for the introduction of new syntax, this feature has little practical importance for us.

In the area of safety, languages allow for more flexible compile-time checking of programming model semantics than are possible with a library. For example, MPI communication takes place using void pointers which discard type information, which prevents the programmer from enforcing consistent types across messaging boundaries. Compiler writers for sequential languages spend enormous time and effort on the automated detection of sequential errors: incorrect function signatures, inappropriate use of pointers, the misuse of constant variables, and so on. This kind of detection can be equally useful in the case of parallel-specific errors. In particular, we discuss the detection of errors in `readonly` variables in section 3.3 and the detection of phase violations in multiphase shared arrays in section 5.4. Charj also benefits from the extension of the type system to cover situations like ensuring that the input and output types of a reduction match appropriately. This kind of static checking gives valuable compile-time information to the programmer which cannot be easily duplicated in a library-based approach.

Finally, the use of a language as opposed to a library gives increased opportunity for optimization. While library code itself can be highly optimized and tuned, it generally does not take unoptimized user code and improve it in the way that a compiler can. In some cases, well-designed libraries can in fact deliver the kind of optimization that is usually only associated with compilers (see, for example, the discussion of iterator-based loop tiling in section 6.1), but a compiler offers greater freedom to transform user code and thereby improve it.

2.3 Infrastructure

Given these guiding principles, we can select the existing software infrastructure on which Charj will be built. Our goals of practical utility and compatibility with existing HPC code point in the direction of established

and successful frameworks. This already narrows our options considerably. Given our desire for tight integration with sophisticated parallel services like load balancing and fault tolerance, we also prefer systems that are feature rich. Moreover, systems that have a rich runtime environment for Charj to interface with provide more interesting opportunities for novel optimizations and syntactic improvements.

With these requirements in mind, we have decided to build Charj on the Charm++ adaptive runtime system. Charm is already widely used and Charm applications account for a significant fraction of total usage at many of the largest clusters in the world. It achieves high performance on a variety of platforms, and there is a pre-existing community of Charm programmers to draw upon. This makes it an attractive target for productivity-enhancing efforts relative to less widely-used systems. It also presents significant complexity to a programmer who wishes to make good use of all its features. In addition to basic messaging capabilities, Charm provides functionality for load balancing [18, 19], semi-automated marshalling and unmarshalling of messages, fault tolerance [20, 21], power management [22], use of accelerator architectures [23], control points [24], and many other features.

In addition, multiple programming models already target the Charm runtime [25–28]. Their existence allows for inquiry into techniques for integrating multiple programming models effectively into a single application. Also, Charm already includes an associated translator which generates messaging code from a programmer-provided interface file. This allows us to compare the advantages of a minimalist approach (generating supplemental interface code only and developing the main application code in C++) to a more thoroughgoing approach in which the compiler for the parallel language has access to method bodies and class structure information. Were we deciding on a platform based solely on popularity and ubiquity we would certainly have built on MPI instead, but given comparative richness and complexity of the Charm runtime system, we claim that Charm is a better environment in which to demonstrate the merits of our approach.

Our goals also influenced our choice of compiler construction tools. Charj is built using the ANTLR LL(*) parser generator [29], discussed further in chapter 4. Its LL(*) parsing algorithm allows for straightforward definition of the language grammar. ANTLR uses a common notation for specifying the language lexer, parser, and abstract syntax tree (AST) traversals,

which substantially simplifies the process of writing the compiler. ANTLR provides a domain-specific language for recognizing and modifying AST subtrees, which we use for simple program transformations and recognition operations. ANTLR also provides us the freedom to build explicit representations of the program outside of its infrastructure, which we use to for more complex analysis.

CHAPTER 3

THE CHARJ LANGUAGE

This chapter describes the Charj programming environment and its relationship to the Charm runtime system. It describes Charj program semantics, syntax and program structure, and gives simple example programs that demonstrate the advantages of Charj programs over their Charm equivalents in terms of concision, safety, and convenience.

3.1 The Charj Programming Model

Charj programs consist of collections of objects which interact via asynchronous method invocation. These objects are called *chares*. Chares can be collected into *chare arrays* or *groups*, or can stand alone. Each chare has a globally unique identifier called a *proxy*, which can be used by other chares to communicate with it. The programmer addresses chares via proxies, rather than by specifying the processor on which the chare resides. This allows the programmer to delegate responsibility for mapping chares onto physical hardware to the runtime system.

Chare objects are specified much as ordinary objects in C++ or Java would be, in terms of their data members and methods. Chares can also inherit from other chares, as is typical in object-oriented design. However, chares contain one or more remotely invocable methods, known as *entry methods*. These methods can be called using only the chare's proxy, even if the caller

resides in a different address space from the callee. Entry methods can take arguments just as normal functions do. These arguments are serialized by the sender, sent to the receiver in a message, and deserialized and used by the receiver. The constructor of a chare class is also considered an entry method, but rather than being sent to an existing instance of the class, it results in the creation of a new instance.

Chare objects can be part of a collection, in which each element has the same chare type. The most common of these is an indexed chare array. Entry methods can be invoked on individual array elements, on the entire array, or on a section of the array. Arguments to entry methods that are sent to multiple array elements are duplicated (except for special cases where duplication may be avoided as an optimization which does not affect the results of a computation), so that a message is sent for each individual receiving chare. The programmer can also conduct asynchronous reductions over the elements of a chare array. Each array element contributes one or more data elements to the reduction, specifying a reducing function and a callback to be called with the result data. The values are combined using the reducing function, and the result is delivered using the specified callback. One special case of chare collections is the group, which is a collection where each physical processor is home to exactly one member of the group. Groups are commonly used to implement application services such as caching and IO.

In a typical application, each processor core will be home to multiple chares. Generally in the Charj programming model, the programmer addresses only individual cores and does not directly program at the level of a multicore node. Throughout this dissertation, when we refer to a processor we mean a single core of a possibly multicore processor node, unless otherwise specified. Messages received by that processor correspond to an entry method invocation on one of the chares located there. Because there may be many such messages outstanding on a processor at any given time, a per-processor scheduler maintains a queue of pending entry method invocations to be processed. The scheduler selects a queue entry and invokes the specified method on the target object using the provided data arguments. The method then runs non-preemptively (and may spawn new entry method invocations of its own). When the method completes, control returns to the scheduler. The scheduler is not guaranteed to use any particular queueing policy, and in-order receipt of messages is not guaranteed. Because application control

flow is driven by the receipt of messages, we refer to this as a *message-driven* programming model.

The message-driven programming model has several important features that make it suitable for large-scale parallel applications. First, it provides a natural way of overlapping communication and computation. Because messages are sent asynchronously, chares do not block execution while waiting to receive data. Instead, the scheduler can select other available messages to process, so that a processor will only go idle if no messages for any of its chares are available. Second, it allows for runtime control of features that would otherwise have to be tightly integrated into application-level code. For example, consider adaptive load balancing. In a model where the programmer addresses application components by their location, the application logic must be explicitly aware of any dynamic movement of those components. In Charj's model, application logic can be more effectively decoupled from the physical location of components, giving us the opportunity to more effectively integrate features like adaptive load balancing and checkpointing into the language itself. The message-driven execution model also effectively supports multi-paradigm parallel applications. As long as each paradigm can be expressed in terms of asynchronous remote method invocations, the code from many distinct paradigms can coexist, mediated by the scheduler. This avoids partitioning of hardware resources between program modules or inefficient time partitioning where an application cannot use multiple models concurrently. We make use of this capability to effectively support a variety of programming model within Charj, as detailed in Chapter 5.

3.2 Charj Syntax

In designing a language targeted at the Charm runtime, we were guided by the principle that new syntax must match the underlying programming model and must always provide a concrete benefit that justifies its inclusion in the language. It is our goal to minimize the time and effort required for a programmer to learn Charj, and to make Charj programs look familiar to anyone acquainted with Charm. To this end, we have adopted a simple Java-like base syntax for serial language constructs and added a small number of new language keywords to support Charm-specific constructs like readonly

variables, entry methods, and chare arrays. Invocations of remote methods and proxies for remote objects are marked with a '@' sigil that allows the programmer to easily distinguish between local and remote operations. Our overarching goal is designing Charj syntax is to make familiar constructs and operations look familiar while drawing attention to Charj-specific features in a consistent and logical way.

A full grammar for the Charj language is provided in Appendix A. In the following sections we highlight language constructs that embody key components of the programming model or which enable particular improvements compared to C++-based Charm++ applications.

3.2.1 Charj Keywords

Several Charj keywords exist primarily to denote the varieties of message-driven entities that are central to Charj programs but which have no direct corresponding concept in other parallel programming models such as MPI and OpenMP. Foremost among these are the keywords for declaring the parallel objects described in section 3.1: `chare`, `group`, `nodegroup`, and `chare_array`. The simplest of these is `chare`, which indicates a parallel object with no particular relationship to other chares in the program or to the hardware on which the application is run.

Whereas in a Charm++ application the programmer creates a normal C++ class and identifies that class as a chare in a separate interface (.ci) file, in Charj chares are declared and defined in the same way that classes and other user-defined data types are, simply using the `chare` keyword instead of `class`. Similarly, programmers can specify parallel collections of objects that are mapped one per physical processing element (groups), or one per physical node (nodegroups) using the `group` and `nodegroup` keywords.

While single chares and per-node and per-processor groups of chares suffice to express many application communication patterns effectively, it is often convenient to create a collection of chares whose elements have a predetermined relationship to one another defined by a characteristic index object associated with each chare element of the collection. These more general indexed collections of chares can be defined using the `chare_array` keyword, which takes an optional dimension argument that specifies the dimensional-

ity of the array's index set. Chare arrays provide a flexible way of creating collections of chares with a well-defined relationship between one another.

Entry Methods

Charj introduces another set of keywords that specify the behavior of the methods of a chare class. First and most important is `entry`, which indicates that a method is remotely invocable via proxy objects. Any attempt to call a non-entry method via a proxy results in a compile-time error. However, entry methods can still be invoked locally in the usual way. The entry keyword is used in the declaration of a function, and comes after any visibility specifiers such as "public" or "private" and before the return type of the function. It is mutually exclusive with the "static" keyword, which indicates that a method belongs to the class as a whole rather than to any particular instance. This is because entry methods are inherently concerned with the particular place where the object corresponding to a proxy resides. Since classes as a whole do not reside in any one location, remote invocation of class methods has no obvious meaning and is disallowed.

Threaded Entry Methods

Any entry method can be designated as a `threaded` method. Threaded methods execute in their own user-level non-preemptible threads. This allows threaded methods to execute blocking operations and return control to the runtime scheduler, which will re-enqueue the blocked method and perform other pending work before resuming the thread. This allows the use of blocking operations in Charj code.

Generally, it is undesirable to make the programmer explicitly specify that a method needs its own thread. If the programmer does not use the threaded keyword on a method that blocks, it results in a runtime error, and errors of this sort are among the problems that Charj aims at ameliorating. However, since it is possible for the programmer to invoke arbitrary code from an entry method and the Charj compiler has no way of determining whether or not that code might block, it is impossible to be sure at compile time whether or not a method needs to be threaded.

One possible solution is to simply make all entry methods threaded. We

rejected this option because threading imposes some extra overhead, and one of our foremost design principles is to avoid any mandatory performance penalties in favor of highly optimizable code. However, this doesn't mean that the programmer is stuck identifying all methods that could potentially block by hand. In practice, most methods that require their own thread need it because they use one of several common runtime features. For example, any method that uses a Multiphase Shared Array (see section 5.4) which changes phase must be threaded. For common cases like this, we can build knowledge into the compiler indicating that particular function calls require that the containing entry method be threaded.

To identify entry methods which must be threaded, we first create a table of expressions which are known to potentially invoke blocking operations. These are typically the invocation of top-level functions, such as the `CthYield()` function which explicitly blocks the current thread and yields control to the scheduler, or methods of known datatypes, such as phase-change functions of the aforementioned multiphase shared arrays. Any of these expressions can be identified in the program's AST using tree pattern matching as described in chapter 4, and the method containing the expression is marked as potentially blocking. Then all callers of that method are also marked potentially blocking, continuing recursively until all potential callers have been marked. We are left with a set of methods known to be potentially blocking (although they may not ever block in actual practice).

Armed with this knowledge, we have two potential courses of action. We can either automatically promote all potentially blocking methods to be threaded, or we can check that the programmer has marked all potentially blocking methods as threaded him- or herself and provide warning or error messages if he or she has not. The advantage of the first option is that it automates as much as possible for the programmer. If we can definitely learn that a method should be threaded, why should we require the programmer to provide that information redundantly? However, consistency argues for the second approach. We must allow the programmer to explicitly specify that a method is threaded to accommodate the invocation of external code not visible to the Charj compiler, which suggests that methods which are not marked "threaded" are indeed not threaded. Automatically threading potentially blocking methods without requiring the use of the "threaded" keyword also makes the threading behavior of the application more opaque

Listing 3.1: Charj source for a generic Node chare class, with one threaded entry method and one local method. All relevant data is located together in a single file.

```
1 // Charj source file (.cj)
2 chare Node {
3     entry Node() {...}
4     threaded entry void receiveData(Data d) {...}
5     void sendData() {...}
6 }
```

to the programmer and increases the difficulty of identifying places in the application which can potentially block.

Since blocking mid-method goes against the normal operating assumptions of a Charj application and provides opportunities for synchronization errors, identifying these places may be relevant when debugging an application. For these reasons, we simply notify the programmer when a potentially blocking method is not marked threaded, rather than promoting the method to its own thread behind the scenes.

The compiler's knowledge about potentially blocking operations can also be used in the opposite direction. Rather than just verifying that the programmer has correctly marked potentially blocking methods as such, it could also identify methods which have been marked as threaded but which contain no potentially blocking calls. This may happen due to code refactoring in which blocking calls are relocated from one threaded method to another or simply due to conservative practices on the part of the programmer. In either case, the compiler can notify the programmer to eliminate the unnecessary overhead caused by threading a method which has no need for its own thread.

Sample Code

To summarize and clarify the relationship between Charj language constructs and their Charm++ equivalents, we present a brief example of the high-level structure of Charj code for a generic chare class called `Node`, with a threaded entry method `receiveData` and a local method `sendData`. Listing 3.1 presents the Charj definitions for such a class, and listing 3.4 gives a Charm++ equivalent.

Listing 3.2: Charm++ equivalent code to the Charj code in listing 3.4. The same information and program constructs are present, but are split across multiple files without any unifying syntax. This listing gives the Charm++ header file (.h).

```
// Charm++ header file (.h)
class Node {
    Node();
    void sendData();
    void receiveData(Data d);
};
```

Listing 3.3: The Charm++ implementation file (.cc).

```
Node::Node() {...}
void Node::receiveData(Data d) {...}
void Node::sendData() {...}
```

Listing 3.4: The Charm++ interface file (.ci).

```
chare Node {
    entry Node();
    entry [threaded] void receiveData(Data d);
};
```

In Charj, chares are declared in the same way as serial classes, but using the `chare` keyword instead of `class`. Entry methods and threaded entry methods are indicated by the use of the corresponding keywords in the method declaration. The declarations and definitions are all grouped together in a common source file, typically with file extension `.cj`.

In Charm++ applications, chares are declared by creating standard C++ classes and identifying them as chares in a separate interface (.ci) file. Listing 3.4 provides a Charm++ equivalent to the Charj code in listing 3.1. There is a direct correspondence of program constructs between the two listings, but the Charj version benefits from consolidating all relevant program information into a single file with a unified syntax, while the Charm++ version splits this data into separate header, implementation, and interface definition files, significantly increasing the size of the code and requiring the programmer to deal with non-local information when working with any one of those files.

Readonly Variables

Applications commonly have need for data which is not known until after the program is running, but which remains unchanged over the life of the program once it is calculated at startup. Typically this data might include proxies to important application chares and program parameters which are read out of configuration files or command line arguments. It is convenient to make this data globally available, but in many parallel programming models the means of providing this data are unnecessarily complex and error-prone.

For example, consider an MPI application that needs to make several parameters from a configuration file available to all ranks. First, to distribute the data, one might use `MPI_Broadcast` to send the variables to all ranks. However, this approach requires either a separate call for each variable to be sent. In fact, if any of the data is of a user-defined type that is not contiguous in memory, it will require even more than that. Particularly if there is a lot of data to share, the distribution of data requires a large number of mostly redundant broadcast calls. Alternatively, the programmer could manually pack the variables into a single buffer and then unpack them on the receiving side. This can increase efficiency by reducing the number of messages sent and received, but introduces more complexity and new opportunities for bugs at the point where buffers are packed and unpacked.

Furthermore, if the programmer forgets to broadcast one of the variables, an uninitialized value will be used, potentially creating bugs. Once the data has been received, the programmer must ensure that the application never assigns to any of the broadcast variables, or else the values held by each rank will no longer be in agreement. This is an important semantic restriction on the program that is not communicated anywhere within the program text and which is invisible to the compiler. The desired behavior is similar to that provided by the "const" keyword, but because the variables must be assigned to during the initialization phase, const variables can't be used without the use of casting tricks.

To address this common need in Charj, we provide "readonly" variables. A readonly variable is declared in the top-level scope using the `readonly` modifier keyword. Readonly variables have special assignment behavior. They can be assigned to freely during the startup phase of the program, in the main chare's constructor. At that time, configuration files can be read, proxies

generated, and so on. When the constructor finishes, all readonly variables are broadcast and made available on every processor. At that time, they become read-only variables, and any assignment to them is an error. This provides increased convenience for the programmer in that they do not have to explicitly broadcast each piece of readonly data. It also provides increased safety by guaranteeing that readonly values remain identical on each processor and are never overwritten.

Readonly variables are not original to Charj. They were first implemented for the Charm runtime system. However, the addition of compiler support in Charj allows for much greater safety and usability of readonly variables. Consider the key property of a readonly variable: after the program's startup phase is complete, the only access allowed to such a variable is read access. In the original Charm implementation of readonly types, this restriction is completely unenforceable. Because the user's application is simply C++, which has no notion of readonly types, the semantic restrictions on readonly variables are up to the programmer to enforce. The Charm++ runtime system does provide a valuable service to the programmer by automating the broadcast of readonly variables at the end of the initialization phase. However, just in terms of safety and enforcement of programming model semantics, this state of affairs is little different from the MPI situation in which the programmer must simply be careful not to overwrite global data and receives no specific help from the system. The Charm++ manual [30] simply states "The current Charm++ translator cannot prevent assignments to read-only variables. The user must make sure that no assignments occur in the program." In fact, it is common to see variable declarations in Charm++ applications annotated with a comment indicating that the variable is readonly, since otherwise that information is available on in the interface file, and the variable cannot be made const.

In contrast, the Charj compiler is aware of readonly types and their semantics. Since the user specifies all readonly variables and the program's startup phase is well-defined by the main chare's constructor, the compiler can verify both that every readonly variable has been assigned to before the end of the startup phase and that no readonly variable is assigned to after the startup phase concludes. Thus, in Charj the semantics of readonly types are directly enforced, whereas in other programming models or in the base Charm model with no compiler support, the burden of ensuring correctness falls only on

the programmer, who receives little or no help from the compiler.

It is important to note that this analysis is not precise, in the sense that there are programs which can never assign to a readonly variable outside of the initialization phase, but which will nevertheless be flagged by the Charj compiler as problematic. Consider, for instance, a function which takes a boolean variable as an argument, and in its body assigns to a readonly variable if and only if that variable is true. If this function is only ever called with a true argument during the initialization phase, then the program is correct. However, in general it is not possible to prove this condition at compile time, and the compiler will conservatively warn the programmer about the assignment. The programmer is then free to evaluate the function in question using their independent knowledge of the program and determine whether or not the assignment in question represents a bug.

Proxy Objects

Proxies are local representatives of remote objects. They consist of a unique identifier that the runtime system can use to local the object in question. A proxy to an object of type T has type "proxy to T," which is roughly equivalent to "pointer to T" with the restriction that a proxy can only be used to invoke entry methods on its referent, and not, for example, to access its member variables or invoke other methods. In the same way that the syntax `T*` indicates a pointer to T, `T@` denotes a proxy to T. Entry methods are also invoked using the `@` operator (in contrast to `->` for pointers). The use of a separate operator for proxies and remote invocation serves to clearly delineate remote objects and operations in application code.

Applications can also make use of proxies to collections of chares, including chare arrays, groups, and nodegroups. Proxies to chare collections are also denoted by a `@`, but messages sent through them are sent to the entire collection. Alternatively, messages can be sent to a single element of a chare array by indexing it.

Proxies allow for a clear expression of the program's parallel structure in a way that can be understood by the compiler. In particular, for any message, the compiler can determine the type of the receiver, the signature of the entry method being invoked, and the types of all arguments to that method. This allows for a significant degree of static checking to be done at compile time.

The compiler verifies that messages are only sent through proxy objects, that the proxies involved expose the intended entry methods, and that the entry methods in question take the appropriate arguments. Compare this to an MPI-style application, where messages are sent to processors rather than objects, and message payloads are all untyped memory buffers. In a well-written program, the programmer's intentions may be clear, and the parallel structure of the application may be readily apparent. However, there is little opportunity for the static detection of programmer errors, and the type system is effectively non-existent for the purposes of checking communication between nodes.

Collectives

Collectives are one of the building blocks that parallel applications are constructed from. Collectives in Charj largely take the form of operations on chare arrays. Broadcasts are handled in much the same way as point-to-point entry method invocations via proxy: an invocation made using a proxy to a chare array (rather than using one of the indexed members of that array) indicates a broadcast to all array members. The type checking described in the previous section applies equally well to broadcasts over chare arrays.

Now, consider reduction operations. The nature of a reduction operation guarantees that the inputs share a common type, that the result of the reduction shares the same type, and that the types of the arguments and result of the reducing operation are also of that type.

Consider the functions used to contribute to a reduction in Charm or MPI, as shown in listing 3.5. The input and output types are unspecified, and there is no guarantee that the types accepted by the reduction operation matches the type of the contributed data. The need to support a wide variety of input types and reduction functions, including user-defined data types and reducers, precludes library designers from effectively encoding the type rules of reductions into their API.

However, by extending knowledge of programming model semantics into the compiler, we can use the type system to catch errors that are not detected by a library approach. The Charj reduction function (also shown in listing 3.5) can verify that the relevant types all match, eliminating the possibility for a reduction operation that doesn't match the contributed data or

Listing 3.5: Function prototypes for reductions on an array of data, in MPI, Charm, and Charj.

```
// MPI Reduction
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
// Charm reduction
void contribute(int nBytes, void* data,
                CkReduction::reducerType type, CkCallback cb);
// Charj reduction
void contribute(Array<T> data, Reducer reducer, Callback cb);
```

contributed data of mismatched types.

Charj provides an even more pronounced improvement versus Charm++ in the case of custom reduction operations. These are reductions in which the data items are of a user-defined type with its own reduction operation. Sample code for supporting custom reductions for a hypothetical `MyType` type, with its own `reduce` function, is given in listing 3.6. Considering that the definition of the type in question and its reduction function are both omitted, the size and complexity of the implementation are notable.

The programmer must engage in non-trivial memory management of runtime data structures associated with reduction trees, and must arrange to register the custom reduction function with the runtime at startup. Beyond the code provided here, the `register_my_reducer` function which adds the custom reduction to the list of reductions that the runtime system knows about must be specified in the Charm++ interface file as an "initcall" function, meaning that it will be executed by the runtime at startup before the main application is started.

In contrast, the equivalent Charj custom reduction code in listing 3.7 is quite brief. Charj custom reducers have an implicit `accum` variable which is used to accumulate new values via the `reduce` method. Reduction registration and runtime reduction message handling code equivalent to the Charm listing are produced from this definition by the Charj compiler, thereby substantially reducing both the length and the complexity of the Charj implementation.

Listing 3.6: Charm++ implementation of a custom reducer for the type `MyType`, which has its own `reduce` function already defined elsewhere. The requirements for explicit handling of system reduction messages and registration of the reduction function with the runtime at startup add significant complexity to the implementation.

```
CkReductionMsg* reduceMyType(int nMsg, CkReductionMsg** msgs)
{
    MyType* accum = new MyType();
    for (int i=0; i<nMsg; ++i) {
        MyType* x;
        PUP::fromMem p(msgs[i]->getData());
        p | *x;
        accum->reduce(x);
    }
    return CkReductionMsg::buildNew(sizeof(MyType), accum);
}

CkReduction::reducerType _my_reducer_type;
void register_my_reducer(void)
{
    _my_reducer_type =
      CkReduction::addReducer(reduceMyType);
}
```

Listing 3.7: Charj implementation of a custom reducer equivalent to the Charm++ code in listing 3.6. Function registration with the runtime and handling of system reduction messages is handled transparently by code generated by the Charj compiler.

```
reducer<MyType> my_reducer {
    my_reducer() { accum = new MyType(); }
    reduce(MyType x) { accum.reduce(x); }
}
```

Generics and Sequential Arrays

While the primary focus of Charj is on expressing parallelism, some of its features are aimed primarily at producing effective serial code. Our goal is to provide the tools necessary for efficient, concise serial code that integrates seamlessly with the parallel-specific features of Charj while maintaining familiar syntax.

One example is the implementation of generic types in Charj. Generic types are important for re-usability and are widely used in both Java and C++. However, their implementations are very different. C++ generics are built on a full template metaprogramming system, which is complex and sophisticated enough that it is Turing complete in itself [31]. In contrast, Java generics work via type erasure and include no metaprogramming facilities. In Charj we need to support generic types, but the complexity of C++ template metaprogramming is a poor fit for Charj's focus on simplicity, particularly in serial code. However, C++ templates have a key performance advantage over Java's type erasure approach, which depends on universal inheritance from the Object class. While recent advances in Java compilers have ameliorated this problem [32], this work falls outside the scope of what we can reasonably include in the Charj compiler.

As a result of these constraints, in Charj we have adopted a generic system whose syntax is substantially similar to Java's, but whose implementation is based on C++ templates. This approach provides the straightforward syntax of Java without sacrificing performance to boxing and unboxing of primitive types, allowing high-performance generics to be used in computational kernels.

The most widely used generic type in Charj is the array. The Array type in Charj denotes the sequential container used within the scope of a single chare object. Arrays in Charj are one of its largest departures from C++-based Charm++ applications. The C and C++ approach, in which array elements are accessed through arithmetic on raw pointers, causes several issues that we wish to avoid. It prevents reliable array bounds-checking, increasing the difficulty of debugging. It makes points-to compiler analysis more difficult by conflating pointers and arrays. It translates poorly to two dimensions and higher, relying on convention to establish layout and, in the case of arrays of arrays, gives up memory locality in exchange for convenient syntax.

Although Java eliminates many of the safety and elegance problems of the C++ approach, typical Java array implementations offer extremely poor performance on HPC workloads, and extensive sophisticated optimizations are required to achieve good performance [33].

To address these problems, we introduce our own generic array type in Charj, simply named Array. It is built into the compiler, and translates to a templatized C++ class implementing the relevant features. Array accesses can be bounds-checked for debugging purposes or left unchecked for maximum performance based on compile-time options. The user can select from row-major, column-major, or block-cyclic data distributions, and redistribute data on the fly as needed.

Charj also provides syntax for specifying ranges over arrays, which allows clear and concise expression of looping constructs. These ranges can also be used to extract contiguous sub-regions of arrays, treating them as independent entities that can be processed without incurring copy overhead.

3.3 Comparing Charm Applications with Charj Applications

It is often the case in discussions of programming languages that any syntax becomes the foremost issue and semantics are neglected. Indeed, according to Wadler's Law[1],

> In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position in the following list:
>
> 1. Semantics
> 2. Syntax
> 3. Lexical syntax
> 4. Lexical syntax of comments

More seriously, it does seem that language discussion is often tightly focused on syntax, perhaps because the syntax is the most obvious feature of

[1] http://www.haskell.org/haskellwiki/Wadlers_Law

any new language. However, many of the most tedious and most error-prone programming tasks in parallel computing have nothing to do with syntax, but are rather matters of semantics.

By incorporating knowledge of the Charm programming model's semantics, Charj greatly simplifies the creation of message-driven applications as compared to a C++ program targeting the Charm runtime. The C++ application must specify type and visibility information for remotely invocable functions and global read only data via an interface file, which the Charm translator uses to generate wrapper code for sending and receiving data. This separates important semantic information about remotely invocable functions from the implementation of those functions, both needlessly duplicating data and making it more difficult for the programmer to get a comprehensive view of the way an application works. A C++ application developer must also be very careful about the semantics of runtime system constructs. For example, Charm provides "readonly" variables which can be assigned only at program startup. There is no facility for enforcing this rule, however, since the application code which accesses these variables is standard C++. Charj resolves these problems simply by eliminating the need for external interface specifications and understanding the semantics of readonly variables, which allows the compiler to enforce their access rules with appropriate error messages.

This basic language and infrastructure serve as the foundation for our work to demonstrate our hypothesis. Even without adding analysis or optimization, this language already provides important productivity benefits relative to using Charm as a C++ library. It eliminates the need for separate interface files which specify which methods may be invoked remotely by cleanly integrating this information into the main body of the program. Standard Charm programs are split into implementation code, headers, and interface files, producing redundancy that can lead to simple errors and inconsistencies. By consolidating the information from these files, Charj presents a unified view of the program that is more concise and can be understood more quickly.

3.4 Example Application

To illustrate the use of Charj in the context of a real program and to highlight differences between equivalent Charj and Charm++ implementations, we present a simple tree-based computation that calculates the Nth Fibonacci number, $fib(N)$. The program consists of a driver main chare class named `Main` (lines 3-17 of listing 3.8 in the Charj version, and lines 4-7 of listing 3.9, lines 3-10 of listing 3.10, and lines 4-16 of listing 3.11 in the Charm++ version. The driver reads from the command line to determine which Fibonacci number to compute, then creates a `Fib` chare to perform the actual computation. The program terminates when the `Fib` chare invokes the driver's `done` method.

The actual computation of the Fibonacci number is performed recursively by the `Fib` class. To compute the $fib(N)$, as long as N is greater than a given threshold value, two new `Fib` chares are spawned, one to calculate $fib(N-1)$ and one to calculate $fib(N-2)$. When they have finished their own computations, they pass their partial results to their parent via the `passUp` method. The parent waits for responses from both children before passing up its own value in turn. The threshold value acts as grainsize control for the application and limits the number of new chares which are spawned.

Listing 3.8: Charj implementation of a simple tree-based Fibonacci application.

```
1  readonly Main@ main;
2
3  public mainchare Main {
4      int n;
5
6      public entry Main(CkArgMsg m) {
7          if (m.argc < 2) n = 16;
8          else n = atoi(m.argv[1]);
9          main = thisProxy;
10         Fib@ fib = new Fib@(true, n, thishandle);
11     }
12
13     public entry void done(int value) {
14         CkPrintf("Fib(%d) = %d\n", n, value);
15         CkExit();
16     }
```

```
}

public chare Fib {
    Fib@ parent;
    boolean root;
    int n;
    int partialResult;
    int pendingChildren;
    const int threshold = 16;

    private int seq_fib(int n) {
        if (n < 2) return n;
        return seq_fib(n-1) + seq_fib(n-2);
    }

    public entry Fib(boolean root_, int n_, Fib@ parent_) {
        n = n_;
        root = root_;
        parent = parent_;

        if (n <= threshold) {
            partialResult = seq_fib(n);
            passUp();
        } else {
            Fib@ child1 = new Fib@(false, n-1, thisProxy);
            Fib@ child2 = new Fib@(false, n-2, thisProxy);
            partialResult = 0;
            pendingChildren = 2;
        }
    }

    public entry void gather(int value) {
        partialResult += value;
        if (--pendingChildren == 0) passUp();
    }

    public void passUp() {
        if (root) main@done(partialResult);
        else parent@gather(partialResult);
        delete this;
    }
}
```

Listing 3.9: Charm++ interface file for the simple Fibonacci application.

```
mainmodule pgm {
    readonly CProxy_Main main;

    mainchare Main {
        entry Main();
        entry void done(int value);
    };

    chare Fib {
        entry fib(bool root_, int n_, CProxy_fib parent_);
        entry void gather(int value);
    };
};
```

Listing 3.10: Charm++ header file for the simple Fibonacci application.

```
#include "pgm.decl.h"

class Main : public CBase_Main
{
    public:
        int n;
        Main(CkMigrateMessage *m) {}
        Main(CkArgMsg *m);
        void done(int value);
};

class Fib : public CBase_Fib
{
    private:
        int n;
        int partialResult;
        int pendingChildren;
        bool parent;
        CProxy_fib parent;
        int seq_fib(int n);
    public:
        Fib(CkMigrateMessage *m) {}
        Fib(bool root_, int n_, CProxy_fib parent_);
        void gather(int value);
        void passUp();
};
```

Listing 3.11: Charm++ implementation file for the simple Fibonacci application.

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

Main::Main(CkArgMsg* m)
{
    if(m->argc < 2) n = 16;
    else n = atoi(m->argv[1]);
    main = thisProxy;
    CProxy_Fib::ckNew(true, n, thishandle);
}

void Main::done(int value)
{
    CkPrintf("Fib(%d) = %d\n", n, value);
    CkExit();
}

Fib::Fib(bool root_, int n_, CProxy_Fib parent_)
{
    root = root_;
    n = n_;
    parent = parent_;

    if (n < THRESHOLD) {
        result = seqFib(n);
        passUp();
    } else {
        CProxy_Fib::ckNew(false, n-1, thishandle);
        CProxy_Fib::ckNew(false, n-2, thishandle);
        partialResult = 0;
        pendingChildren = 2;
    }
}

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

void Fib::gather(int value) {
    partialResult += value;
```

```
42      if (--pendingChildren == 0) passUp();
43  }
44
45  void Fib::passUp()
46  {
47      if (root) main.done(partialResult);
48      else parent.gather(partialResult);
49      delete this;
50  }
51
52  #include "pgm.def.h"
```

Despite the brief and simple nature of this example code, many important differences between Charm++ and Charj are apparent in these listings. First, and perhaps most importantly, the Charm++ version is significantly more verbose. Despite the identical structure of the two implementations and the fact that no particularly space-saving Charj features such as custom reductions are used in the code, the Charj version weighs in at 58 lines, compared to the Charm++ version, which takes 92 lines spread over three files. The Charm++ version is over 1.5 times as long, mostly due to replication of information across the interface file, header file, and implementation file.

The bodies of the functions which perform the actual work are largely identical between versions. The biggest exceptions to this rule are in the use of parallel-specific features, specifically the invocation of entry methods, which are marked by the `@` symbol in Charj as opposed to a period in Charm++, and the creation of new `Fib` chare objects, which are created via `new Fib@` in Charj, and via `CProxy_Fib::ckNew` in Charm++. The similarity in serial code serves to lower the barrier to entry for new Charj programmers, while the new parallel-specific syntax calls attention to explicitly parallel operations and distinguishes them from operations on local objects.

3.5 Summary

In this chapter we have described the structure of the Charj language and its relationship to the Charm++ runtime system and programming model. Charj aims to ease the process of writing message-driven applications by providing language constructs well-suited to the task and more tightly inte-

grating the language with key programming model concepts. This approach allows for greater concision and simplicity, while also facilitating greater type safety and opening up the possibility for better feedback to the programmer in the form of meaningful warnings and error messages. It also creates the opportunity for compile-time optimizations that are not possible with a library-based approach.

CHAPTER 4

THE CHARJ COMPILER

In order to demonstrate the value of a compiler to a rich runtime system, one must have a compiler. For reasons outlined in section 2, we have elected to build our own compiler and associated infrastructure rather than adopting the software infrastructure from a pre-existing compiler project.

In this chapter, we describe the overall architecture of the Charj compiler and the steps by which Charj source code is turned into an executable for the target architecture. This includes the tokenization, lexing, and parsing of the input program, the construction of an abstract syntax tree (AST) and symbol table, semantic analysis, optimization, and code generation. The specific optimizations performed by the compiler will be deferred to chapter 6, and here we will only describe the high-level structure of the compiler that supports these specific optimizations.

4.1 Software Ecosystem

One of our primary goals with Charj is to create a tool that is actually useful in practice for creating programs based on the Charm++ runtime system. In order to accomplish this goal, we must allow the programmer to make use of the preexisting suite of tools that exist to support Charm++ programs, while adding new Charj-specific tools that interact well with the existing codebase.

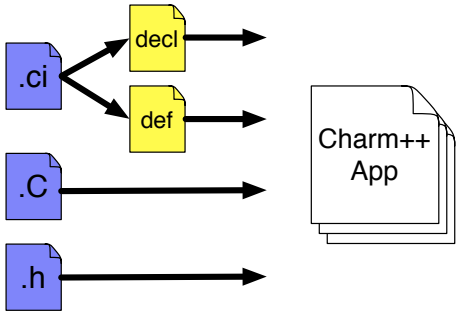
As discussed in chapter 3, Charm++ programs are largely composed of

C++ code, with an accompanying interface (.ci) file that specifies information about parallel-specific features of the code. The Charm++ software distribution includes a translator, charmxi, which can read an interface file and produce stub code that ties the programmer's application code to the runtime system. The translator produces two output files: the declarations file (.decl.h), which contains forward declarations for all Charm++-specific functions and variables, and the definitions file (.def.h), which contains their implementations. These generated files are then included in the user's C++ implementation, along with any needed C++ headers, and from that point on the process of producing a functioning application binary is identical to that of standard C++, with the caveat that the binary must be linked against the Charm++ runtime libraries.

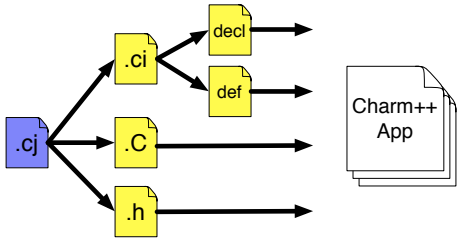
The process of creating a binary from the C++ source code that results from the combination of the user's own code and the output of the charmxi translator is not specific to Charm++. However, this process can become quite involved, given that the user must specify the include path for the Charm++ system headers, the path to the Charm++ libraries and any libraries that they depend upon, and provide the appropriate flags to the linker. These flags may vary significantly depending on the particular compiler and compiler version being used and the location of system libraries on the machine where compilation occurs. To mitigate this problem and simplify the toolchain needed by Charm++ programmers, the standard distribution of Charm++ includes a wrapper script, charmc, which handles many of the details of the translation, compilation, and linking process.

One advantage of using Charm++ as the basis for Charj is the ability to make use of the significant institutional support for Charm++. Default Charm++ installations are commonly provided on supercomputers, and the engineering effort required to make the Charm++ software environment work effectively across a wide variety of hardware and software configurations has already been done. By piggybacking on the existing Charm++ infrastructure, we avoid a substantial effort that is not directly tied to our research goals.

In order to effectively integrate with Charm++, we provide tools to aid the programmer in going from a Charj program (possibly interacting with or partially composed from Charm++ code) to a functional application, without giving up access to the features provided by charmc. The core Charj



(a) Compilation process for a Charm++ application.



(b) Compilation process for a Charj application.

Figure 4.1: In a Charm++ application, the programmer specifies an interface (.ci) file that accompanies the C++ code that forms the bulk of their application and specifies type signatures and visibility information about remotely invocable functions. A corresponding Charj program integrates this information directly into the application, and the Charj compiler generates code targeting the Charm runtime.

compiler is a Java application described in detail in the following sections. It takes Charj source files as input and outputs C++ code and Charm interface definitions suitable for compilation by charmc, as shown in figure 4.1. The Charj compiler accepts a number of optional command-line arguments that control features such as the verbosity of its diagnostic output and the level of warning and error messages produced.

In order to simplify the compilation process for end users, we provide a wrapper script called charjc. This wrapper accepts as arguments the union of legal arguments to the Charj compiler and legal arguments to charmc. It invokes the Charj compiler on the input source files, passing all Charj options through. It then takes the Charm++ interface and C++ code output of the Charj compiler and invokes charmc on them, applying the remaining charmc command-line flags. The output of this process includes both the source code output of the Charj compiler and the binary output obtained from charmc. This output can be linked directly with the Charm libraries and Charj runtime. By making the output of the Charj compiler as close as possible to a normal Charm program, we make it easier to integrate Charj code into existing Charm code and vice-versa, while also giving the programmer an easy way of inspecting the outcome of the Charj compilation process. Although the Charj compiler and the charjc wrapper script that handles argument passing and invoking charmc on the output of the Charj compiler

are distinct entities, for brevity we use the name of the wrapper script which invokes the Charj compiler, charjc, synonymously with the Charj compiler itself in places where this distinction is not important.

4.2 Compiler Architecture

The Charj compiler is a Java application composed of several modules. The main components of the compiler are the parser, the abstract syntax tree (AST) handler, the symbol table and associated symbol definitions, and the code generator. The parsing, AST manipulation, and code generation are all implemented using ANTLR [29], which dictated the choice of Java for the application as a whole.

The compiler driver is essentially a Java wrapper around the core compiler functionality. The driver parses command-line arguments, reads input, constructs the appropriate ANTLR objects to first construct an AST, and then to perform passes over that AST and generate code, optionally outputting debugging information about the current state of the AST on each pass. It also manages the creation of the output source files. Before passing the input file to the ANTLR parser, the driver first preprocesses the source using the `cpp` preprocessing tool. While the initial design of the Charj language did not include the use of preprocessor macros, in practice we found that the need for conditional compilation of application code in HPC applications was so widespread that support for this conditional compilation was necessary. The use of `cpp` allows this conditional compilation in a way that is already familiar to Charj programmers and eases the porting of existing C and C++ codes that use conditional compilation extensively, usually to enable and disable architecture-specific performance optimizations.

4.2.1 Generating an AST

ANTLR (short for ANother Tool for Language Recognition) provides domain-specific languages for specifying language grammars and constructing and manipulating ASTs. From the specification, ANTLR creates code to tokenize and lex Charj source input and construct an AST. ANTLR also provides a language, called *filter grammars*, for recognizing and modifying AST

Listing 4.1: ANTLR grammar rules for Structured Dagger statements. Each rule consists of a list of alternative patterns with associated AST outputs.

```
1  sdagTrigger
2      : IDENT ('['! expression ']'!)? formalParameterList
3      ;
4
5  sdagStatement
6      : OVERLAP block
7          -> ^(OVERLAP block)
8      | WHEN (sdagTrigger (',' sdagTrigger)*)? block
9          -> ^(WHEN sdagTrigger* block)
10     ;
```

subtrees. Charjc is a multi-pass compiler, and each pass is implemented as a series of operations on subtrees identified via ANTLR filter grammars. Simple code transformations and recognition operations are implemented directly within these filter grammars, but for more complex tasks we construct an explicit representation of the program's control flow graph (CFG) and operate directly on that data structure in Java rather than relying on ANTLR's domain specific language.

Listing 4.1 illustrates rules from the ANTLR specification for Charj's grammar, specifically for Structured Dagger statements described in section 5.3. Each rule consists of a list of alternatives. Each alternative is composed of *tokens*, such as `IDENT` and `WHEN`, *literals*, such as '`[`' and '`,`', and other rules, such as `expression` and `formalParameterList`. ANTLR allows supports extended Backus-Naur Form (EBNF) notation [34] for denoting alternation, repetition, optional elements, and so on within the alternatives.

Each alternative is associated with an output AST. The default is to create a tree whose nodes are the elements of the alternative, with the first element as the root and each subsequent element is a child. Elements suffixed with a '!' are excluded from the resulting AST. In the example listing, the `sdagTrigger` rule uses this method of AST creation.

Alternatively, AST outputs can be specified explicitly. ANTLR's AST notation has the following form:

```
1  ^(root child1 child2 ... childN)
```

AST outputs can be explicitly constructed using this notation by affixing a `->` symbol to the alternative, followed by the desired result AST, as is done for the alternatives for the `sdagStatement` rule in the example listing.

The specification of grammar rules is made simpler by the flexibility of ANTLR's `LL(*)` parsing algorithm, which eliminates or mitigates many common problems experienced in the use of common LR-based parsers such as YACC [35]. `LL(*)` parsing is a generalization of LL(k) parsing featuring arbitrary lookahead, which eliminates the need for the grammar writer to determine the correct value of `k`, and allows for grammars that are not `LL(k)` for any fixed `k`.

4.2.2 Semantic Analysis and Optimization

Once the input is parsed and the AST is created, the compiler makes several passes over the AST prior to code generation. The purpose of these passes is to analyze the AST and extract information that is useful either for providing more effective warnings and error messages to the programmer, or to aid in the process of code generation.

These passes are written in terms of ANTLR *tree pattern matchers* [36], which allow the programmer to specify the structure of AST subtrees of interest and actions to be performed when those subtrees are encountered, in either a top-down or a bottom-up traversal of the tree. This avoids the need to describe the entire AST structure for each pass, while still allowing the use of descriptive ANTLR syntax for describing subtrees. In addition, it abstracts the details of the tree traversal operation and the process of identifying AST substructures away from the action to be performed once those substructures are encountered.

For example, consider listing 4.2, which shows a portion of a tree pattern matcher which identifies class variables that require initializers in the class's constructor and/or inclusion in the class's generated pack/unpack (PUP) routine (see section 6.2 for a description of PUP methods and PUP-related optimizations in Charj).

Each top-level rule in the listing describes a tree structure using ANTLR syntax. The rules for describing trees in this way are more relaxed than for the full language grammar, since rules here match entire families of subtree. Most notably, it allows the use of the '`.`' and '`*`' operators to denote an arbitrary tree node, and an arbitrary repetition of the preceding element, respectively. So, for example, the pattern `^(TYPE .*)` would match any sub-

tree whose root is a `TYPE` node. Then, following each rule, the programmer can specify a block of Java code to be executed when the rule is matched, or a rewrite rule which specifies a transformation of the matched subtree, or both. If the AST is modified during a traversal, it is re-walked until an entire traversal takes place with no AST modifications.

Additionally, there are two special top-level rules in a tree pattern matcher: `topdown` and `bottomup`. These rules simply list the patterns that can be matched when walking the tree from top to bottom and from bottom to top, respectively. These can be used to track the current location of the traversal within the tree. In the example listing, the rules `enterMethod` and `exitMethod` are only used to track whether or not the traversal is currently within a class's method when it matches the `varDeclaration` rule, because local variable declarations within a method do not need class-level initialization or inclusion in PUP routines.

The `varDeclaration` rule simply matches the AST structure for a variable declaration, including any initialization expression. The associated code action for this rule first verifies that the declaration occurs within a class, but not within the definition of one of its methods. Then, if the declaration includes an initializer, it is added to its class's initialization list, using the AST associated with the initialization expression. The variable is also added to the list of class variables used for generating the PUP function, and a distinction is made between proxy types and non-proxy types for the sake of simpler processing later on.

Tree pattern matchers of this type are widely used in Charj to perform tasks such as type resolution, symbol table population, and identification of places in the program that are candidates for optimization or possible sites of errors.

4.2.3 Code Generation

When our optimizations and analysis are complete, we output C++ code and Charm interface code which is compiled against the Charm API. ANTLR integrates tightly with the StringTemplate template engine. StringTemplate provides a way to produce structured text directly from the AST structure without coupling the AST structure to the format of the output [37] . One

Listing 4.2: ANTLR filter grammar rules used for identifying class variables that need to be initialized and packed/unpacked. Context in the form of the symbol of the current class is maintained in the `enterClass` and `exitClass` rules, and all AST subtrees that match the pattern associated with variable declarations

```
1  topdown : enterClass | enterMethod | varDeclaration;
2  bottomup : exitClass | exitMethod
3
4  enterClass :
5          ^(TYPE .*) {
6              currentClass = $IDENT.def.sym;
7          };
8
9  exitClass :
10         ^(TYPE ...) {
11             currentClass = null;
12         };
13
14 enterMethod :
15         ^((FUNCTION_DECL | ENTRY_FUNCTION_DECL) .*) {
16             inMethod = true;
17         };
18
19 exitMethod :
20         ^((FUNCTION_DECL | ENTRY_FUNCTION_DECL) .*) {
21             inMethod = false;
22         };
23
24 varDeclaration :
25         ^(VAR_DECLARATOR ^(IDENT .*) (expr=.)? ) {
26             if (!inMethod && currentClass != null) {
27                 if ($expr != null) {
28                     currentClass.initializers.add(
29                         new VariableInitializer($expr, $IDENT));
30                 }
31
32                 currentClass.varsToPup.add($IDENT);
33                 if (!($IDENT.symbolType instanceof ProxyType ||
34                       $IDENT.symbolType instanceof ProxySectionType))
35                     currentClass.pupInitializers.add(
36                         new VariableInitializer($expr, $IDENT));
37             }
38         };
```

of our goals in code generation is to produce output that can be read and readily understood by the programmer. Because the overall structure of a Charj application and its Charm++ equivalent are generally quite close, it is straightforward for the programmer to look at the generated output and find a correspondence between the generated code and their input Charj code. This is an important quality to maintain in order to maximize the programmer's ability to debug Charj applications, particularly when Charj code is being integrated with existing Charm++ modules or components.

In order to maximize the similarity of the generated code to the input code, we preserve the original identifier names, and use meaningful names for generated variables whenever possible. For example, local variables in methods that contain Structured Dagger constructs must be promoted to class variables (see section 5.3). In order to avoid name conflicts, the promoted variable names must be mangled to ensure their uniqueness. Rather than using meaningless random names, we combine the original variable name, the name of the method in which it is declared, and a number indicating the scope within that method where that variable was declared. The scope indicator is necessary because two variables with different types but the same name can be declared in different blocks within the same function. So, for example, the variable `iteration` declared at the top level of the `calculate` method would appear in its mangled form as `_iteration_calculate_1`.

From the final program AST, we produce three different output source files: a C++ header file, a C++ implementation file, and a Charm++ interface file. To achieve this, we write a generic function for walking the AST and invoking a set of StringTemplate templates to produce output. Each template is associated with a particular type of AST node. We produce the three output files by supplying three different sets of templates to the generic code generation function.

For example, consider the templates in listing 4.3, which demonstrates slightly simplified templates associated with the declaration of entry methods. The templates each take a list of arguments, each of which is either a symbol from the symbol table or a template associated with some subtree of the AST rooted at the current node. So, for example, the "`modifiers`" argument to the template is itself the template associated with the list of keyword modifiers for this method, such as `public`, `static`, or `threaded`, and the "`block`" argument is the template associated with the body of the

Listing 4.3: Simplified StringTemplate templates associated with an entry method declaration for each of C++ header, C++ implementation, and Charm++ interface output targets. The arguments to the template are themselves the templates associated with subtrees of the AST rooted at the entry method declaration, or, in the case of classSym and methodSym, symbols representing the current class and method.

```
// Header output template
entryMethodDecl_h(classSym, methodSym, modifiers,
                  type, id, params, block) ::=
<<
<modifiers><type> <id><params>;
>>

// Interface output template
entryMethodDecl_ci(classSym, methodSym, modifiers,
                   type, id, params, block) ::=
<<
<modifiers><type> <id><params>;
>>

// Implementation output template
entryMethodDecl_cc(classSym, methodSym, modifiers,
                   type, id, params, block) ::=
<<
<if(block)>
<modifiers><type> <classSym.Name>::<id><params>
{
    <if(methodSym.isTraced)>
    int _charj_method_trace_timer = CkWallTimer();
    #endif
    <endif>

    <block>

    <if(methodSym.isTraced)>
    traceUserBracketEvent(<methodSym.traceID>,
           _charj_method_trace_timer, CkWallTimer());
    <endif>
}
<endif>
>>
```

method.

When generating output for the header or interface files, the body template will return an empty string because the function body does not appear in those files, while in the implementation file it will expand to the whole body of the method in question. The use of symbol arguments allows us to access information stored in the symbol table to make decisions about what to output. In the example given, tracing code will be inserted in the method body if that method's symbol indicates that it should be traced. The unique id used for tracing is also stored in the symbol data structure.

4.3 Summary

The Charj compiler is an essential component of our research agenda. It gives concrete form to the Charj language, and creates the opportunity to explore a wide variety of optimizations specific to Charj programs. It is fair to say that the Charj compiler does not contain any novel new technology which will advance the state of the art in compiler research. In fact, we designed it to rely on well-known and thoroughly tested techniques. However, this software acts as a solid foundation from which to explore the possibilities of productivity-enhancing techniques specific to message-driven applications.

CHAPTER 5

EMBEDDING DIVERSE PROGRAMMING MODELS

As parallel applications grow larger and more complex, it becomes less and less feasible to write an entire application using a single programming model. In a large application with multiple constituent modules, no one paradigm is necessarily suitable for writing the entire application. While it has in the past been common practice to produce applications that exclusively use message passing, or global arrays, or actors, or any of a number of other models, this approach is unnecessarily limiting, and may force the programmer to choose a compromise model that is only mostly suitable and force the entire application to use that model. In particular, if one wishes to make use of parallel modules which encompass some task-specific parallel algorithm, it is difficult to require that the module use the same programming model as the rest of the application without sacrificing programmer productivity via longer development time, lower maintainability, and generally inelegant code.

There are several benefits to multi-model parallel applications. They enable freer choice of libraries and modules and encourage code re-use. They allow a "right tool for the right job" approach in which, for example, an array-based model can be used for array-intensive parallel code while a model specialized for tree-structured parallel computations can be used where trees are the central data structure. They also allow the use of incomplete models, which are models that are not capable of expressing arbitrary parallel inter-

actions but which in return are able to provide increased safety guarantees and more elegant notation to programmers.

One powerful technique for making use of multiple programming models in a single application is for all the models to target a common parallel runtime system. The runtime system can mediate communication between modules and schedule code belonging to different models in an intelligent way because it has access to and control over the entire state of the application. This allows for the minimization of compatibility layers between models and the potential for overlapping execution of code belonging to different models.

In this work, we take advantage of the ability for the Charj compiler to be aware of multiple programming models that can be used together in a single application. Charj programs can then provide much greater integration between models via improved static checking and model-specific optimizations.

5.1 Related Work

Multi-model (or multi-paradigm) programming and ability of programming languages to accommodate that style is a topic that extends far beyond the confines of parallel computing. For example, C++ is often referred to as a multi-paradigm language, in that it supports programming with an imperative style, object-oriented programming, and template-based meta- and functional programming. For the purposes of this discussion, however, we limit ourselves to those systems which allow multiple programming paradigms which are specifically parallel in nature. These systems can be roughly categorized as either multi-paradigm parallel languages, parallel programming model extensions, interoperability frameworks, and runtime systems which unify multiple programming models.

Before we begin a discussion of the particular programming models that we have embedded in Charj, we should first make more explicit what we mean by "embedding" in this context, as this term can denote several distinct techniques for incorporating multiple models within a program or programming environment.

In some cases, supporting embedded programming models means that the language provides support for users or library writers to provide their own custom syntax which closely integrates with the parent language, and which

allows syntax for domain-specific models which were not envisioned by the original language developers to be cleanly integrated in the basic model after the fact.

One example of this use is"Rakefile" syntax in the Ruby programming language, which provides constructs for managing software build and deployment processes similar to the common Unix "make" utility. Rakefiles are valid Ruby programs which incorporate this embedded special syntax to simplify the process of describing dependencies, build environments, and so on [38,39]. Similar techniques have been applied in other high-level languages such as Scala [40] and Haskell [41].

This practice of users embedding syntax in a parent language is perhaps best exemplified by the Lips macro system, which allows the programmer to specify sophisticated operations on the Lisp s-expressions that make up their program. This allows the creation of new control structures and so on at the level of libraries or application code. The extensive use of such constructs in Lisp programs is made possible by the fact that Lisp programs are essentially explicit representations of their own syntax tree, with no intermediate syntax. As a result, Lisp is a very effective platform for embedding new syntax to support domain specific languages or otherwise extend core language functionality [42].

Another use of "embedding" in the context of programming models is to describe the use of preprocessing tools which transform an input file that is a mix of the parent model and the embedded model and output a pure program that consists only of code in the parent model. This allows the developer of the embedded model to piggyback on the infrastructure provided by the parent model and potentially reduce the barrier to entry for potential users of the embedded model. For example, the MetaBorg system allows the user to embed domain-specific languages in the Java language to support such tasks as user interface specification and XML generation [43, 44], and Lua-ML allows the embedding of Lua code within ML applications [45].

However, these uses are not what we mean to indicate by the term "embedding." Rather, we take embedding to mean that we incorporate syntax and other program constructs from other special-purpose programming models or extensions to the base message-driven Charj programming model directly within Charj programs in a fashion that can be understood by the Charj compiler and all targeted to a common runtime system. Whereas in the original

Charm++ system, these models would require either their own translator support or implementation as a C++ library, in Charj we can directly incorporate them as first-class citizens, making them possible targets for static checking and optimizations, as well as simpler, more direct syntax. These models typically target a particular class of parallel interactions, such as disciplined access to global arrays, or the interaction between host hardware and accelerator hardware in software which runs on hybrid systems. By embedding these models within Charj, we make it easier for programmers to make use of the advantages they provide in their particular problem domains while maintaining the advantage of a unified Charj infrastructure.

We also avoid the need to embed models which would benefit from custom syntax in a language which supports custom syntax only weakly. While we discussed the embedding of custom syntax in Ruby and Lisp above, these languages are not widely used in HPC application development. In that arena one is much more likely to encounter programs written in C, C++, or Fortran. C and Fortran provide essentially no support for incorporating special-purpose syntax into their base syntaxes. C++ provides greater opportunities, particularly through its template metaprogramming facilities. Indeed, the multiphase shared arrays programming model which we embed in Charj and which is discussed further in section 5.4 was originally implemented as a C++ library, and later revised to improve the static checking provided to the programmer by leveraging the C++ type system. However, C++ provides only weak embedding facilities compared to languages like Lisp and Ruby, and greatly constrains the syntax and language constructs that a model implementer can feasibly provide. As we discuss in the subsequent section, embedding this model directly in Charj provides a variety of benefits that are not available to implementations that are constrained to use only the facilities provided by C++.

5.2 Supporting Multiple Programming Models

We believe that the use of multiple programming models can provide significant productivity benefits to the programmer, particularly in the context of large applications with many linked components. In Charj, we attempt to support a variety of programming models and notations within the base Charj

programming environment, all operating on a common runtime system. In the following sections, we will describe the models that Charj supports, the ways in which these models are useful to HPC programmers, and the benefits that accrue from integrating them into the Charj programming environment.

5.3 Structured Dagger

Structured dagger (SDAG) [27, 46, 47] addresses a common need in parallel message-driven applications to effectively coordinate the sequence of execution between the methods of communicating objects. SDAG facilitates this process by providing a clear expression of the flow of control within an object while maintaining the ability to adaptively overlap communication and computation.

In the basic message-driven model the body of a remotely invokable method contains serial code which does not block, and when data must be received from a remote entity, callbacks are typically used. This approach suffers from the non-local nature of program control flow. Because each entry method is directly invoked by the scheduler, and the receiver of a message may respond with any of a variety of return messages or none at all, the natural pattern of messages that are passed between objects over the course of an application's lifetime is not immediately obvious from the code, and may require significant interpretation by skilled programmers to discover. In addition, the programmer has no built-in way of describing common interaction patterns such as "proceed when I have received n messages of type t", as when an object waits for its neighboring objects before continuing a computation. The advantage of this scheme is that it allows the runtime system to adaptively overlap communication with computation and supports data-dependent communication patterns, but in some cases it does so at the expense of program clarity.

Structured dagger is a coordination language aimed at clarifying control flow in message-driven applications and supporting common idioms needed in applications which depend on asynchronous method invocation, without sacrificing the associated benefits of overlap of communication and computation. It defines several constructs that allow the programmer to express message-driven control flow within the context of a single method.

The most important of these constructs is the "`when`" block. A when block

specifies dependence between the arrival of a particular type of message and the execution of a given block of code. Syntactically, the when block has a name, an argument list, and an associated block of code, much like a function definition. The name is used by other elements of the application to trigger the when, the argument list specifies the type of data expected by the when, and the block of code is executed when the message is received. A when block may contain multiple pairs of names and arguments, in which case the block is only executed after all the expected incoming messages have been received.

SDAG also defines an "`overlap`" block, which specifies that its constituent components can be concurrently enabled and executed in any order. The actual order of execution will depend on the order in which triggering messages are received. The overlap block only completes once all of its components complete.

Additionally, SDAG supplies a "`forall`" keyword, which acts like a `for` loop in which all the iterations can be overlapped with one another.

SDAG defines several additional constructs to denote serially executable C++ code and to allow conditional execution and looping, but these constructs primarily exist to allow SDAG code to coexist with serial code rather than to enable new parallel-specific functionality. Specifically, "`atomic`" blocks indicate that their contents are simply C++ code that contains no SDAG constructs. This keyword is an artifact of SDAG's initial implementation, and does not persist as a keyword or program concept in Charj. SDAG also defines the control-flow constructs "`if`," "`for`," and "`while`", which are all semantically equivalent to their C counterparts.

Listing 5.1 illustrates the use of SDAG to express the main loop of a simple Jacobi relaxation application with a one dimensional data decomposition. In each iteration, every chare sends a strip of boundary elements to each of its neighbors. Once the chare has received strips from each of its neighbors (via `getStripFromLeft` and `getStripFromRight`), the actual stencil computation can be performed via `doStencil`. The reception of the left and right boundary regions is overlapped via the `overlap` construct.

In this example code, the sending of left and right neighbor information is elided via the use of the `sendStrips` method. Within this method, each chare would invoke the `getStripFromLeft` and `getStripFromRight` entry methods on its neighbors using proxy objects, providing the necessary infor-

Listing 5.1: A simple SDAG function illustrating the use of `overlap` and `when` statements in the context of an iterative Jacobi stencil application with a 1-D decomposition.

```
1  entry void jacobi()
2  {
3      for (int i=0; i<N; ++i) {
4          sendStrips();
5          overlap {
6              when getStripFromLeft(Strip s) {
7                  processStripFromLeft(s);
8              }
9              when getStripFromRight(Strip s) {
10                 processStripFromRight(s);
11             }
12         }
13         doStencil();
14     }
15 }
```

mation to them via the `Strip` argument, which simply encapsulates the strip of array elements that constitutes a boundary between array chunks. These entry methods are not defined explicitly by the programmer, but rather are implicitly defined by their use as SDAG triggers within the `overlap` statement. Their use in the `when` statements creates a synthetic entry method which receives the expected data and invokes the code indicated by the trigger once the data has all arrived. In this case, the indicated code is the `processStripFromLeft` or `processStripFromRight` method. These methods, which are also elided for brevity, simply extract the data necessary for the eventual stencil operation and store it in a place where it can be accessed conveniently in the eventaul `doStencil` call.

The equivalent C++ message-driven version of the Jacobi code in listing 5.1 is provided in listing 5.2. The differences between these code listings illustrate a few of the advantages that the use of SDAG can bring to message-driven applications.

The first and most obvious advantage of the SDAG implementation is its brevity. While the actual work involved in the Jacobi computation is not included for the sake of clarity and brevity, the entire top-level structure of the application is contained in a single fifteen line function. The equivalent message-driven version spans five different functions and uses more than double the lines of code.

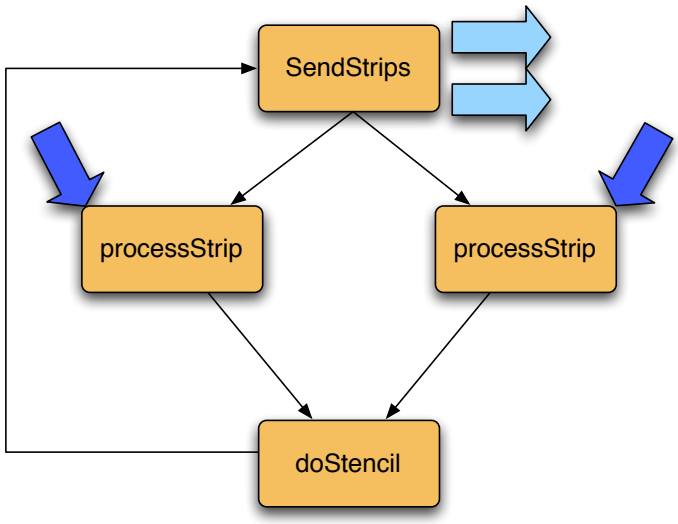


Figure 5.1: The control flow associated with the jacobi code in listings 5.1 and 5.2, from the perspective of a single chare containing one array chunk. Two messages are send out to neighbors initially in the `sendStrips` function. Then incoming messages from neighbors trigger left and right `processStrip` methods. Once these have both completed, the stencil computation can take place, at which point the process is repeated in the next iteration.

However, merely comparing the length of the two listings understates the advantage in clarity that SDAG provides. Adding additional functions does not only increase the length of the code. It also increases the mental burden on the programmer, because the nature of the interactions between these functions is never explicit in the code. To determine the path of execution that will occur when the code is run, the programmer must reason carefully about the chain of messages and function calls that will occur. In addition, it is not clear if these functions might be called from other code elsewhere in the application. Furthermore, the functions do not all correspond to natural units of work. The fact that the loop calculations are now split across the three functions `jacobi`, `mainLoop`, and `checkOverlapCompletion` obscures the programmer's intent and forces the reader to jump between various points in the program with no obvious connection in order to determine the overall control flow of the application. Reasoning about code becomes much more difficult because the desired semantics of the function are not made explicit as they are in the SDAG version.

In addition, the message-driven version of the code suffers from an increased need for state variables. The overlap between receiving the left and

Listing 5.2: The message-driven equivalent of the SDAG Jacobi function in listing 5.1. The simple control flow expressed in the SDAG loop is broken into several interacting functions.

```
1  entry void jacobi()
2  {
3      i = 0;
4      mainLoop();
5  }
6
7  void mainLoop()
8  {
9      leftStripReceived = rightStripReceived = false;
10     if (i < N) {
11         sendStrips();
12     }
13 }
14
15 entry void getStripFromLeft(Strip s)
16 {
17     processStripFromLeft(s);
18     leftStripReceived = true;
19     checkOverlapCompletion();
20 }
21
22 entry void getStripFromRight(Strip s)
23 {
24     processStripFromRight(s);
25     rightStripReceived = true;
26     checkOverlapCompletion();
27 }
28
29 void checkOverlapCompletion()
30 {
31     if (leftStripReceived && rightStripReceived) {
32         doStencil();
33         ++i;
34         mainLoop();
35     }
36 }
```

right boundary regions is accomplished transparently in the SDAG code, but requires the addition of two state variables to determine when both sides have been received. While the additional overhead is small in this case, in larger and more complicated functions, the number of state variables required to track control flow can become onerous. Here, SDAG does not reduce the computational or storage overhead associated with dependency tracking. However, it does effectively hide this complexity from the programmer and present a clear view of the control flow of the application without the need for exposing the state variables needed to implement it in a message-driven system.

5.3.1 Implementing SDAG

In Charm++, SDAG is implemented as a system of complex C++ macros, partially produced through the Charm++ translator. This allows it to introduce new syntactic constructs while keeping it tightly bound to the Charm/C++ application and obviating the need for significant SDAG-specific compilation tools. However, this approach entails significant compromises in exchange for the convenience of avoiding a full language definition and compiler infrastructure.

SDAG neatly illustrates the difficulty of building new parallel programming models to interoperate with existing C++ applications without any compiler support. SDAG defines a small set of new keywords which can be used to specify the high-level communication structure of message-driven code, avoiding some of the problems of non-local control flow and hidden dependencies that can make message-driven applications difficult to follow. However, its implementation as a macro system added on to C++-based Charm code is very limiting, despite the fact that the Charm translator provides it with some code generation capabilities.

Atomic Blocks

The most obvious limitation of SDAG is that while its constructs include C++ expressions and blocks of arbitrary C++ code, the SDAG infrastructure has no way of parsing C++, and adding general-purpose C++ parsing is notoriously complicated. As a result, C++ blocks inside SDAG constructs

must be enclosed in an "`atomic`" block which renders the contents of the block invisible to the SDAG translator. This process is error-prone because the translator must assume that all code in the atomic block can be correctly parsed by a C++ compiler later, and if this is not the case the resulting error messages can be confusing. It is also fragile, because the translator relies on being able to match curly braces to determine where each `atomic` block ends. If the programmer erroneously omits a curly brace inside a block, the resulting error message is very confusing. The translator must also go to efforts to detect whether or not braces within a block are inside a comment or not. In addition, SDAG has no way of verifying that the code contained in these blocks obeys the rule that code in an `atomic` block invokes no parallel coordination operations, nor does the SDAG translator parse the expressions that it uses for conditional and looping constructs, eliminating any possibility for optimizations or warnings and errors based on these expressions. Outside the context of a parser that understands block contents, the process of translating SDAG code is messy and fragile.

Beyond this lack, SDAG also suffers from its implementation as a macro system. Once the SDAG code is generated by the translator, the user must insert multiple SDAG-specific macros into any class that uses SDAG methods. These macros then expand into the orchestration code that comprises SDAG. The error messages if the programmer forgets these macros are necessarily opaque and unhelpful to programmers who have not experienced them before, and contribute to the difficulty of using SDAG. In addition to these inconveniences, the way in which SDAG code is generated prevents an SDAG method from invoking other SDAG methods, which is a serious problem for anyone who wishes to use SDAG as a significant part of a real application. Furthermore, the way that SDAG methods are split apart by the translator prevents the use of local variables that span multiple blocks.

One fundamental limitation of combining a simple SDAG translator with C++ applications is poor integration of sequential code with SDAG code. C++ is far too difficult for a simple translator to parse, but it would be extremely limiting to completely segregate SDAG constructs that indicate the conditions under which messages should be sent and actions should be taken from the C++ code that actually implements those actions. To get around this problem, the SDAG translator introduces the "atomic" construct, consisting of the keyword `atomic` followed by a block of sequential code enclosed

in curly braces. When the SDAG translator encounters an atomic block, it treats the contents as a black box to be inserted into generated code, and does not attempt to the inner C++. This allows intermixing of serial code with SDAG code without complicating the translator. However, it does so by imposing an additional semantic burden on the programmer, forcing them to insert additional syntax that has no bearing on meaning of the code in question. This problem is exacerbated by the fact that SDAG uses several common control flow constructs in its own grammar to allow the programmer to express application messaging behavior. In particular, SDAG methods may contain "if" statements and "for" and "while" loops that operate identically to their C++ equivalents, even to the point of allowing arbitrary C++ expressions in the conditionals and loop initializers and updaters, but which are nevertheless considered SDAG code which should not be enclosed in an atomic block. This semantic mismatch is confusing to programmers who are not familiar with the implementation of the SDAG translator, and the need for atomic block specifiers is an annoyance even to experienced SDAG programmers.

In Charj, SDAG constructs coexist with the sequential portions of the language, with no arbitrary separation between them. The compiler can infer the existence of sequential blocks of code that execute when SDAG triggers fire, and emit correct code based on this knowledge. This eliminates the status of standard control flow constructs like "for" and "if" as quasi-SDAG constructs that have their normal semantic meaning but are used as though they are not part of the normal sequential code in a method.

Initialization and Communication of SDAG Data Structures

Some of the limitations imposed by attempting to graft SDAG onto a C++-based programming environment are not particularly deep from a technical perspective, but nevertheless impose a substantial cognitive burden on the programmer. To integrate code generated by the SDAG translator into the larger C++ application, the programmer inserts special macros and function calls into their code. Specifically, for each class with SDAG methods, the programmer inserts an `SDAG_CODE` macro in the class body to insert the generated code into the class, calls the `__sdag_init()` function in the class's constructor, and calls the `__sdag_pup()` function in the Chare's PUP function

(see section 6.2) to handle serialization and deserialization of SDAG-specific data structures.

Inserting these function calls and macros is not in itself a huge burden on the programmer, but the necessity for these additions degrades programmer productivity in several ways. First, they represent an easily forgotten and somewhat arbitrary additional step that the programmer must remember when coding. They be forgotten initially, but they also represent a continuing maintenance burden. For example, if a Chare does not initially need a PUP function because it is never migrated, then no `_sdag_pup()` call is needed. However if a PUP function later becomes necessary, perhaps to facilitate dynamic load balancing, then the programmer who adds this PUP function must be aware that the class contains SDAG methods and that he or she must therefore insert the appropriate call. The likelihood of errors is increased by the fact that SDAG code must all be placed in an interface file that normally contains only declarations and no method definitions. Since the actual SDAG code is segregated from the bulk of the C++ implementation code, it is more easily overlooked. In the event that one or more of the macros and function calls is forgotten, the resulting errors can be subtle and difficult to track down, particularly for programmers who aren't familiar with the details of SDAG's implementation techniques. Particularly in the case of omitted initialization and pup calls, the errors may manifest themselves only as subtly incorrect data, and bugs may manifest themselves only on certain platforms. In addition, SDAG's macro-based implementation makes it more difficult to debug the serial code within SDAG methods. The SDAG translator does not attempt to parse this code. It simply breaks the sequential blocks within the SDAG method into separate message-driven methods, which are injected into the programmer's Chare class via the `SDAG_code` macro. As a result, the C++ compiler doesn't see this code before it is broken into pieces and inserted via macro. Any compile-time errors in the code will therefore refer to source code that was generated by the SDAG tools, in a method that corresponds to some section of the original SDAG code, but which contains automatically generated message handling code and non-semantic method and variable names. Figuring out the relationship between this generated code and the original SDAG method can be a daunting task for programmers who are not already well-versed in SDAG.

Handling Sequential Code

Fortunately, these problems are not inherent to the SDAG programming model. They are only present as an artifact of the way that the SDAG translator coexists with a C++ application. If the SDAG translator was able to properly understand C++ code and interface with the programmer's classes, these issues could be easily avoided. In Charj, we have no such limitations. The same parser is used to handle sequential Charj code and all SDAG-specific constructs. Therefore all warning and error messages related to the sequential code can be emitted as normal, with reference to their location in the original source file and in their proper context. Furthermore, there is no need for the insertion of macros or SDAG utility functions in separate user code, since in Charj there is no distinction between the compilation of code which contains SDAG constructs and code which does not.

Handling Local Variables

SDAG programming suffers from a variety of small warts and annoyances that stem from its lack of tight integration with the larger C++ application. For example, one significant limitation of of the original SDAG implementation is its lack of support for local variables. Because of the way that the SDAG translator breaks each SDAG function into a series of independent message-driven functions, local variables declared in one part of an SDAG function do not persist in other regions of the function. Even a loop index variable may not be visible within the entirety of the loop body. To get around this restriction, SDAG programs typically promote what would normally be local variables to class variables in the enclosing Chare. This approach has several drawbacks. First and most obviously, it removes what would normally be locally scoped information, moving the information contained in the variable declaration farther away from the point at which that information is useful and exposing that data to wider visibility than is necessary. The increased visibility of what would otherwise be local variables is also a potential source of bugs. If, for example, two SDAG methods in the same Chare happen to both use the loop index variable "i", execution from portions of each SDAG method can be interleaved, giving incorrect results. The programmer must therefore be careful that the semantically local variables needed by each

SDAG method are in fact only used locally.

This problem is addressed in a straightforward way by the Charj compiler. In a Charj program, variables may be declared in the normal way in a function that contains SDAG constructs. The compiler ensures that any accesses to these local variables obey the normal variable scoping rules. However, when generating code, the compiler promotes these variables to become class variables of the enclosing Chare. The names of the variables are mangled with the name of the scope in which they are declared, so that there can be no incorrect aliasing that would lead to unexpected, incorrect behavior. Local variables in SDAG methods do incur more overhead than local variables in other methods, because they require persistent storage in their containing class rather than living entirely on the stack, but because the semantics of SDAG method execution guarantee that the method's stack frame will be torn down and control returned to the scheduler before the method is completed, this limitation is unavoidable.

Although the hoisting of local variables in SDAG methods to class variables is not particularly sophisticated from a compiler analysis perspective, it is emblematic of the things we aim to accomplish with Charj. It provides real utility to the programmer and simplifies SDAG programs without needing to be complicated or sophisticated. Although the ability to declare local variables in SDAG methods is not necessarily a momentous one in terms of its impact on the programmer, the accumulation of small advantages like this can quickly become significant.

Calling SDAG Methods

One important criterion for assessing SDAG's integration into a programming environment is the ease with which SDAG methods can be called. Unfortunately the original implementation of SDAG imposes some unintuitive restrictions on the programmer. Although SDAG methods superficially appear identical to other entry methods, they may contain asynchronous control flow, in that they may block to await expected incoming messages. In order to support this control flow, SDAG must preserve local state across any blocking operations. Furthermore, the SDAG code generation must be aware of all code locations where control may be ceded to the scheduler.

These requirement put some important restrictions on the calling of SDAG

methods in the original translator. First and foremost, the programmer cannot invoke any blocking functions from within an atomic block. Doing so would change the parallel control flow DAG represented by the function in a way that is invisible to the translator and produce incorrect results. Note that since anything within an atomic block is opaque to the SDAG translator, these errors cannot be caught at compile time. Typically they present as corrupted data or race conditions in the user's code. Such errors are particularly time consuming to debug, especially when invoking third-party code with which the programmer may not be intimately familiar.

In practice, the largest inconvenience caused by the inability to call blocking functions from within SDAG methods is the inability to call one SDAG function from the body of another SDAG function. This limitation prevents the programmer from performing common refactoring and code organization tasks and can result in unneeded and unwanted duplication of code, increasing maintenance costs and creating new opportunities for bugs.

One goal of our Charj implementation of SDAG constructs was to ease this limitation on the calling of blocking functions. The Charj compiler's ability to analyze sequential code blocks and do inter-procedural analysis provides us with all the tools needed to determine the correct DAG for any SDAG method. However, the fact that the Charj compiler emits Charm and SDAG code puts limits on what we can achieve without reimplementing the logic of the original SDAG translator within Charj. There is no way of representing such entities as a recursive SDAG function in a way that the SDAG translator can understand.

We therefore narrow our scope to simply allow SDAG functions to invoke other SDAG functions, with the restriction that we do not allow mutual recursion. The Charj compiler observes that one SDAG method is calling another SDAG method, and embeds the body of the callee within the caller, renaming local variables as necessary. This process is repeatedly invoked until all SDAG calls are expanded. The resulting SDAG methods each contain their entire parallel control flow DAG, and can therefore the output of the Charj compiler can be processed by the original SDAG translator. This provides a useful service to the programmer (i.e. increased flexibility in the calling of SDAG methods) without requiring the reimplementation of the SDAG infrastructure within the Charj compiler. There are no theoretical problems involved with allowing arbitrary calls from within SDAG methods,

but in practice doing so would involve substantial implementation effort to bring SDAG infrastructure into the Charj compiler.

Handling "When" Triggers

We can also see signs of SDAG's uneasy integration in the way that `when` triggers are declared. Each trigger is associated with one or more actions, as in `when my_action(int x) atomic { ...action... }`. The names of these actions correspond to methods that are generated by the SDAG translator, so the programmer does not supply definitions for the actions. However, the programmer must still provide declarations for each action type in the interface file. So, for the example given, the programmer would have to declare `entry void my_action(int x);` in the interface file. This looks identical to declarations for which the programmer would be required to also supply a definition, muddling the issue of which interface definitions correspond to actual user code somewhere in the application and which definitions only correspond to generated code. In Charj, we scan all SDAG code to identify triggers without the need for separate declarations. This reduces redundant information and avoids potential confusion on the part of the reader.

The aggregate impact of these differences between the Charj SDAG implementation and the original SDAG translator is a much tighter integration between SDAG code and the rest of an application. This integration can have a substantial qualitative effect on the experience of developing applications that contain SDAG. In fact, one of the most important SDAG-related improvements in Charj is simply the capability to freely add SDAG constructs in any Charj method without the need for relocating them to a different file or marking their definitions with special keywords. The total effect of these changes is to make SDAG much easier and less frustrating to use. These benefits come without any modification of the SDAG feature set and with no degradation of performance. In short, by bringing SDAG into the Charj programming model as a first-class citizen, we provide a much more cohesive and seamless experience to the programmer.

5.4 Multiphase Shared Arrays

Multiphase Shared Arrays (MSA) [25, 28] is a programming model for distributed arrays in a partitioned global address space, where array accesses are governed by a shared sequence of access modes and synchronization points. MSA addresses a common problem faced by shared memory applications: non-deterministic outcomes due to data races. These data races may lead to time-consuming, difficult-to-find bugs, and eliminating them while maintaining high performance is a difficult task.

One problem common to shared memory applications are data races, where concurrent access to globally visible data yields a non-deterministic result. The initial development of MSA was based on the observation that applications that use shared arrays typically do so in phases. Within each phase, all accesses to the array use a single mode, in which data is read to accomplish a particular task, or updated to reflect the results of each thread's work.

MSA provides a formal way of expressing and enforcing this phase structure by requiring that phases be explicitly declared and separated by synchronization points. This allows MSA to provide a guarantee of deterministic behavior and freedom from data races and deadlock. However, it also confines MSA to express only a subset of all possible parallel interactions. It is not a general-purpose parallel programming model, and as such it is only useful as one part of a rich multi-model environment in which it can perform its limited role extremely well while leaving general-purpose parallelism to other models.

5.4.1 The MSA Programming Model

MSA provides an abstraction common to several HPC libraries, languages, and applications: arrays whose elements are simultaneously accessible to multiple client threads of execution, running on distinct processors. These clients are user-level threads, typically many on each processing element (PE), which are tied to their PE unless explicitly migrated by the runtime system or by the programmer. Application code specifies the dimension, type, and extent of an array at the time of its creation, and then distributes a reference to it among client threads. Each element has a particular *home* location, defined by the array's *distribution*, and is accessed through software-managed caches.

By establishing this discipline, MSA usage is inherently deterministic. However, in exchange for this guarantee, the programmer gives up some of the freedom of a completely general-purpose programming model.

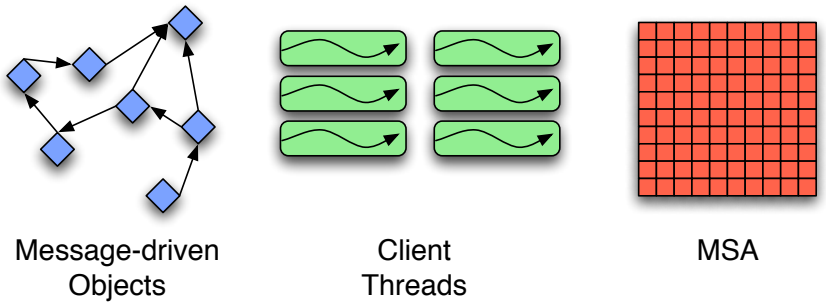
Data Decomposition and Distribution

MSA decomposes arrays not into fixed chunks per PE, but rather into *pages* of a common shape. Developers can vary the shape of the pages to suit applications' needs. For example, a 10×10 array could be broken into ten 10×1-shaped pages, or four 5×5 pages, etc. Thus, the library does not couple the number of pages that make up an array to the number of processors on which an application is running or the number of threads that will operate on that array. If the various parts of a program are *overdecomposed* into sufficiently more pieces than there are processors, the granularity of communication associated with data transfer can be effectively controlled, and the runtime system can flexibly map pages to available hardware resources.

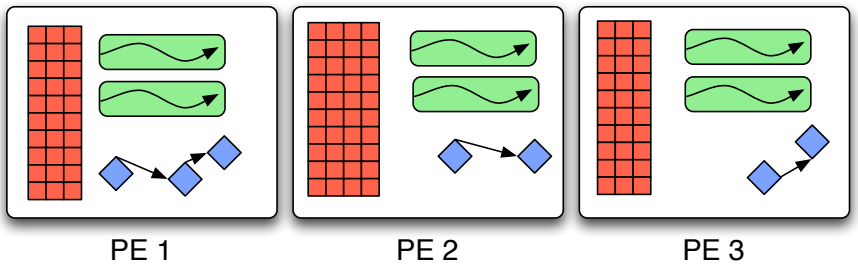
At the simplest, the pages can take a blocked row- or column-major arrangement, with the block shape determined by the library to suit the underlying memory and communications hardware. MSA allows the application programmer to manually specify one of a few simple decompositions, and can be extended to support more complex cases as application needs dictate.

Once the array is split into pages, the pages are distributed among PEs. The page objects are managed by the Charm++ runtime system. Thus, each MSA offers control of the way in which array elements are mapped to pages, and the mapping of pages to PEs. This affords opportunities to tune MSA code for both application and system characteristics. The page objects are initially distributed according to a mapping function, either specified by application code or following the defaults in Charm++. As the program executes, the runtime may redistribute the pages by migrating them to different PEs in order to account for load imbalance, communication locality, system faults, or other concerns. The user view of an MSA program and corresponding mapping by the runtime system are illustrated in figure 5.2.

The drawback of this scheme is high latencies for non-local reads and phase changes. The runtime compensates by overlapping the execution of other local threads with blocking MSA operations. This process is enabled by overdecomposition, so that on each PE there are many threads using the



(a) The user view of an MSA application.



(b) One possible mapping of program entities onto PEs

Figure 5.2: The developer works with MSAs, client threads, and parallel objects without reference to their location, allowing the runtime system to manage the mapping of entities onto physical resources.

MSA. When the active thread blocks, either due to an MSA cache miss, or a non-MSA operation, another thread can be scheduled.

Caching

The runtime library caches data accessed from MSAs. This approach differs from Global Arrays [48], where the user must either allocate and manage buffers for bulk remote array segments or incur remote communication costs for each access. It is more similar to caching schemes in UPC, with the difference that MSA's phase structure places much fewer restrictions on communication optimizations than even UPC's most relaxed memory model [49]. Runtime-managed caching offers several benefits, including simpler application logic, the potential for less memory allocation and copying, sharing of cached data among threads, and consolidating messages from multiple threads.

When an MSA is used by an application, each access checks whether the element in question is present in the local cache. If the data is available, it is

returned and the executing thread continues uninterrupted. The programmer can also make prefetch accesses spanning particular ranges of the array, with subsequent accesses specifying that the programmer has ensured the local availability of the requested element. Bulk operations allow manipulation of an entire section of the array at once, as in Global Arrays.

These prefetch calls can be blocking or non-blocking, as the programmer desires. This scheme naturally lends itself to optimization by a compiler. When static compiler analysis can determine array access patterns, prefetching can be done transparently, improving program performance without intervention by the programmer. This technique is discussed in detail in chapter 6.

When a thread accesses data that is not cached locally, the cache requests it from its home page, then suspends the requesting thread. At this point, messages queued for other threads are delivered. The cache manager receives the home page's response and unblocks the requesting thread. Previous work with MSA [50] has shown that the overhead of caching and associated checks is reasonable, and well-tuned application code can match the performance of equivalent sequential code.

Each PE hosts a cache management object which is responsible for moving remote data to and from that PE. Synchronization work is also coalesced from the computational threads to the cache objects to limit the number of synchronizing entities to the number of PEs in the system. Depending on the mode that a given array is in, the cache managers will treat its data according to different coherence protocols, as the Munin system does [51]. However, the MSA access modes are designed to make cache coherence simple and inexpensive. Accesses never require remote cache invalidations or immediate writeback.

In write-once mode, all writes to remote data can be buffered until the end of the phase, minimizing communication costs. Runtime verification that the write-once guarantee has not been violated takes place within the home objects (see below) when remote writes are committed at the end of the phase. Similarly, accumulations are performed in a local buffer, and the result is consolidated with the remote data during the phase change.

Access Modes and Safety

By limiting programs to a few well-defined access modes and requiring synchronization from all MSA client threads to pass from one mode to another, race conditions within the array are excluded without requiring the programmer to understand a complicated memory model. The access modes MSA provides are suitable for many common parallel access patterns, but it is not clear that these modes are the only ones necessary or suitable to this model. As we extend MSA further, we expect to discover more as we explore a broader set of use cases.

Read-Only Mode: As its name suggests, read-only mode makes the array immutable, permitting reads of any element but writes to none. Remote cache lines can simply be mirrored locally, and discarded at the next synchronization point. In this mode, there are no writes to produce race conditions.

Listing 5.3: A characteristic array access in read-only mode.

```
1 x = a(i, j, k);
```

Write-Once Mode: Since reads are disallowed in this mode, the primary safety concern when threads are allowed to make assignments to the array is the prevention of write-after-write conflicts. We prevent these conflicts by requiring that each element of the array only be assigned by a single thread during any phase in which the array is in write-once mode. This is checked at runtime as cached writes are flushed back to their home locations. Static analysis could allow us to check this condition at compile time for some access patterns and elide the runtime checks when possible.

Listing 5.4: A characteristic array access in write-once mode. The element (i, j, k) cannot be assigned to by multiple threads within one phase, but could potentially be assigned to multiple times by the same thread.

```
1 a(i, j, k) = y;
```

Accumulate Mode: This mode effects a reduction into each element of the array, with each thread potentially making zero, one, or many contributions to any particular element. While it is most natural to think of

accumulation in terms of operations like addition or multiplication, any associative, commutative binary operator can be used in this fashion. One example, used for mesh repartitioning in the ParFUM framework [52], uses set union as the accumulation function. The operator's properties guarantee that the order in which it's applied does not introduce non-deterministic results.

Listing 5.5: A characteristic array access in accumulate mode. Contributions made by a particular thread are cached locally until the next synchronization point, at which point contributions from different threads are accumulated together at each array page's home location.

```
1 a(i, j, k) += z;
```

The use of the various access modes are illustrated in the following Charm++ code snippet that computes a histogram in array `H` from data written into array `A` by different threads:

Array subscripts are set off by parentheses, rather than the more conventional square brackets, so that syntax remains consistent when accessing arrays of dimension greater than one. This is a restriction imposed by C++'s different rules for overloading the subscript (`operator[]`) and call (`operator()`) operators. This restriction does not apply to the Charj implementation of MSA, as discussed in section 5.4.2.

Even when considering only one-dimensional arrays, C++ operator overloading presents a problem. Depending on where it is used, an MSA array access can be either an *lvalue* (that is, a value that can be assigned to), or an `rvalue` (that is, a value that can be assigned, but not assigned to). C++ operator overloading facilities are too restrictive to allow the range of operations that we wish to express through the overloading of the bracket operators.

Synchronization

A shared array moves from one phase to the next when its client threads have all indicated that they have finished accessing it in the current phase, by calling the synchronization method. During synchronization, each cache flushes modified data to its home location and waits for its counterparts

Listing 5.6: A characteristic array access in accumulate mode. Contributions made by a particular thread are cached locally until the next synchronization point, at which point contributions from different threads are accumulated together at each array page's home location.

```
1  A.syncToWrite();
2
3  for (int i = 0; i < N/P; ++i)
4      A(tid + i*(P-1)) = f(x, i);
5
6  A.syncToRead(); // Done writing A; data can now be read
7  H.syncToAccum(); // Get ready to increment entries in H
8
9  for (int i = 0; i < N/P; ++i) {
10     int a = A(i + tid*N/P);
11     H(a) += 1;
12 }
```

on other PEs to do the same. Logically, client threads cannot access the array again until synchronization is complete. In SPMD-style MSA code, this requires that threads explicitly wait for synchronization to complete sometime before any post-synchronization access

5.4.2 Implementing MSA

Because Charj uses the same runtime system as MSA, it is straightforward to make use of MSA within Charj programs. Because the Charj compiler has explicit knowledge of the MSA programming model, it can provide a far better experience for the programmer than the C++ MSA library can.

One area in which this advantage shows itself is in enforcement of MSA's various access modes. Detection of MSA programming model violations is made difficult by the fact that MSA is implemented as a C++ library. In the original implementation of MSA, the access mode of each phase was implicit in the structure of the code. Phase boundaries were delimited by `sync()` calls, but there was no mechanism for determining the intended phase structure of an application aside from comments or a close reading of the code to determine which kinds of accesses are used inside of each phase. The process of determining array phase becomes even more difficult when considering that arrays may be passed into a function from many different places in an application, and the function's signature gives no indication

of the expected array phase. The MSA library performs checks at runtime to ensure that there are no access mode violations, but this process incurs performance overhead for the checking, lengthens the debugging process, and leaves the possibility that unexercised code paths contain MSA access mode errors.

To address these problems, MSA was later redesigned to enforce as many of its access restrictions as possible using the C++ type system. In the modified MSA, all accesses to the array take place through a handle object, and there is a different type of handle for each kind of MSA phase (i.e. there is a read phase handle type, a write phase handle type, an accumulate handle type, et cetera). In addition, the single `sync()` call is replaced with specialized calls that perform synchronization and return a handle of the appropriate type for the ensuing phase, for example `syncToRead()` which performs synchronization to end the previous phase and returns a read handle to be used in the next phase. The interfaces of these handles enforce the access rules of the phase by exposing only allowed operations. So, for example, trying to write a value using a read mode handle will create a type error at compile time. C++ operator overloading is used to support familiar array syntax.

Using the C++ type system does address some of the problems of the original implementation. However, it suffers from problems of its own. Whereas code using the original library could present itself as a straightforward series of operations on a single array variable, the new interface requires a proliferation of handle objects, which are often short-lived and provide little value to the programmer. This problem could be addressed by the use of linear types [53], but this option is precluded by the need to work within the C++ type system. The use of the C++ type system to detect access mode violations is also fairly limited in the types of errors that it can detect at compile time. MSA uses read-only mode, exclusive write mode, and accumulate mode. These modes can all be enforced by careful declaration of MSA operations in the handles (for example the array accessors in read mode are all `const`), but this technique is not easily generalizable to other access modes. To some extent this deficit could be addressed using policy templates and static assertions in the latest C++ standard, but this would complicate the library interface significantly.

Other techniques for enforcing MSA semantics rely on external tools. MSAs high-level semantic conditions could be enforced using a *contract* ap-

proach [54] describing allowable operations. However, the use of contracts in the context of the MSA C++ libary would require an external enforcement tool and the contract conditions would depend on state variables that aren't visible in user code. Some alternative external static analysis tool could also provide an enforcement mechanism, but any such tool would have to do flow-dependent analysis and replicate much of the work of the compiler in a separate application.

However, when implementing MSA as an integrated part of Charj, we can avoid some of the problems inherent in expressing it as a C++ library. We can have the best of both worlds: simple syntax which does not rely on a typed handle approach, combined with compiler enforcement of the MSA safety properties. Simply by observing all accesses to a particular array, the compiler can verify that the accesses during any given phase are consistent, provided that it has a complete view of the lifecycle of the array. In general, this requires a whole-program analysis. However, this excludes the possibility of calling Charj functions which take MSA arguments from C++ code (because the mode of the MSA when it enters the function is unknown). Pragmatically, we issue errors if any inconsistent array accesses are detected at compile time, and warnings if functions are exposed that could potentially be misused by external code unavailable to the compiler.

5.4.3 Static Checking

We perform static analysis to detect violations of MSA's access mode restrictions. Our analysis is similar in concept to the computation of *reaching definitions* [55]. The reaching definitions for a given point in a program are the set of definitions which may have been the point at which a value used at that point was defined in some path of execution. We wish to answer a similar question: which points of synchronization of an array might have started the current phase when an array is accessed. Because each synchronization indicates the nature of the ensuing phase, we can then verify that no synchronization point can lead to an invalid array access without an intermediate point of synchronization that changes the phase of the array.

In computing the synchronizations that reach a given array access, we need concern ourselves only with accesses to MSAs and synchronization operations

on those arrays. For each of these statements S, we compute $in[S]$, the set of incoming reaching synchronizations $out[S]$, the set of outgoing reaching synchronizations, $gen[S]$, the set of synchronizations generated by S, and $kill[S]$, the set of synchronizations killed by S. The governing dataflow equations are the same as in reaching definitions:

$$in[S] = \bigcup_{p \in pred[S]} out[p]$$
$$out[S] = gen[s] \cup (in[S] - kill[S])$$

where $pred[S]$ is the set of predecessors of S. This simply states that the synchronizations incoming to S are those outgoing from its predecessors, and that the outgoing synchronizations are the incoming synchronizations together with synchronizations generated by S and excluding synchronizations killed by S. The gen and $kill$ sets are only modified by synchronization statements. Consider a synchronization statement x on an array a. The rules governing $gen[S]$ and $kill[S]$ for this statement are given by:

$$gen[S] = \{x\}$$
$$kill[S] = X_a - \{x\}$$

where X_a is the set of all synchronizations on array a. Using these definitions we can compute the set of reaching synchronizations for each array access by iteratively applying the dataflow equations until the $in[S]$ and $out[S]$ remain unchanged for each statement.

This analysis is conservative, in that it will detect violations that may not ever occur in actual execution. For example, consider the code in listing 5.7. Whatever the value of the `condition` variable, the array `A` will always be synchronized to write mode before it is written to. However, we do not attempt to determine which branches will be taken during actual execution, and in general it is not possible to identify all cases that would actually be safe in practice as safe. We therefore report potential access mode violations as warnings rather than errors. If the programmer determines that the actual pattern of access is safe, he or she can simply ignore the warning. A future version may support pragmas to identify accesses that are known to be safe to the compiler, allowing the programmer to eliminate warning messages which are known to be spurious.

Listing 5.7: Although the array A will always be in read mode when it is accessed, we will still warn the programmer of a possible MSA access mode violation.

```
A.syncToRead();
if (condition)
    A.syncToWrite();
if (!condition)
    A.syncToWrite();
A[0] = 0;
```

5.4.4 Example Application

To illustrate the difference between the typed handle syntax and the direct access syntax, we provide a sample MSA application which performs a histogramming task. There are two MSAs involved in the computation: a 2D `data` array which is filled with random numbers, and a 1D `bins` array which holds a histogram of the data the first array.

The handle-based application in listing 5.8 and 5.9 consists of a simple `Driver` mainchare which creates worker `Histogram` chares and then waits for results to be delivered via the `done` entry method. The Histogram objects have a single entry method that is invoked to do the main work of the application, which must be threaded to allow MSA operations to block. The additional MSA declarations are needed in order for the correct MSA templates to be instantiated at compile time.

All accesses to the array take place through the typed handles `MSA2D::Read`, `MSA1D::Accum`, etc. Each change of phase requires a new handle instantiation, leading to a proliferation of variables throughout the program. However, these handles ensure that the accesses to the arrays obey the appropriate MSA phase rules, so that, for example, no elements are read out of an array that is in write mode.

Listing 5.8: The interface (.ci) file for the Charm++ histogram application with typed handles.

```
mainmodule histogram
{
    mainchare Driver
    {
        entry void Driver(CkArgMsg*);
        entry void done(CkReductionMsg*);
```

```
    };

    array [1D] Histogram
    {
        entry void Histogram(MSA2D data_, MSA1D bins_);
        entry [threaded] void start();
    };

    // Any MSA templates used in the application must be
    // explicitly instantiated in the interface file.
    group MSA_CacheGroup<int, DefaultEntry<int>,
                         MSA_DEFAULT_ENTRIES_PER_PAGE>;
    array [1D] MSA_PageArray<int, DefaultEntry<int>,
                             MSA_DEFAULT_ENTRIES_PER_PAGE>;
};
```

Listing 5.9: The implementation (.cc) file for the Charm++ histogram application with typed handles.

```
#include "msa/msa.h"

typedef MSA::MSA2D<int, DefaultEntry<int>,
        MSA_DEFAULT_ENTRIES_PER_PAGE, MSA_ROW_MAJOR> MSA2D;
typedef MSA::MSA1D<int, DefaultEntry<int>,
        MSA_DEFAULT_ENTRIES_PER_PAGE> MSA1D;

#include "histogram.decl.h"

const unsigned int ROWS = 2000;
const unsigned int COLS = 2000;
const unsigned int BINS = 10;
const unsigned int MAX_ENTRY = 1000;
unsigned int WORKERS = 10;

class Driver : public CBase_Driver
{
public:
    Driver(CkArgMsg* m)
    {
        // Usage: histogram [number_of_worker_threads]
        if (m->argc > 1) WORKERS=atoi(m->argv[1]);
        delete m;

```

```
25          // Actually build the shared arrays: a 2d array to hold arbitrary
26          // data, and a 1d histogram array.
27          MSA2D data(ROWS, COLS, WORKERS);
28          MSA1D bins(BINS, WORKERS);
29
30          // Create worker threads and start them off.
31          workers = CProxy_Histogram::ckNew(data, bins, WORKERS);
32          workers.ckSetReductionClient(
33              new CkCallback(CkIndex_Driver::done(NULL), thisProxy));
34          workers.start();
35      }
36
37      void done(CkReductionMsg* m)
38      {
39          // When the reduction is complete, everything is ready to exit.
40          CkExit();
41      }
42  };
43
44
45  class Histogram: public CBase_Histogram
46  {
47  public:
48      MSA2D data;
49      MSA1D bins;
50
51      Histogram(const MSA2D& data_, const MSA1D& bins_)
52      : data(data_), bins(bins_) {}
53
54      Histogram(CkMigrateMessage* m) {}
55
56      ~Histogram() {}
57
58      // Note: it's important that start is a threaded entry method
59      // so that the blocking MSA calls work as intended.
60      void start()
61      {
62          data.enroll(WORKERS);
63          bins.enroll(WORKERS);
64
65          // Fill the data array with random numbers.
66          MSA2D::Write wd = data.getInitialWrite();
67          if (thisIndex == 0) fill_array(wd);
```

```
68
69          // Fill the histogram bins: read from the data array and
70          // accumulate to the histogram array.
71          MSA2D::Read rd = wd.syncToRead();
72          MSA1D::Accum ab = bins.getInitialAccum();
73          fill_bins(ab, rd);
74
75          // Print the histogram.
76          MSA1D::Read rb = ab.syncToRead();
77          if (thisIndex == 0) print_array(rb);
78
79          // Contribute to Driver::done to terminate the program.
80          contribute();
81      }
82
83      void fill_array(MSA2D::Write& w)
84      {
85          // Just let one thread fill the whole data array
86          // with random entries to be histogrammed.
87          //
88          // Note: this is potentially a very inefficient access
89          // pattern, especially if the MSA doesn't fit into
90          // memory, but it can be convenient.
91          for (unsigned int r = 0; r < data.getRows(); r++) {
92              for (unsigned int c = 0; c < data.getCols(); c++) {
93                  w.set(r, c) = random() % MAX_ENTRY;
94              }
95          }
96      }
97
98      void fill_bins(MSA1D::Accum& b, MSA2D::Read& d)
99      {
100         // Determine the range of the data array that this
101         // worker should read from.
102         unsigned int range = ROWS / WORKERS;
103         unsigned int min_row = thisIndex * range;
104         unsigned int max_row = (thisIndex + 1) * range;
105
106         // Count the entries that belong to each bin and accumulate
107         // counts into the bins.
108         for (unsigned int r = min_row; r < max_row; r++) {
109             for (unsigned int c = 0; c < data.getCols(); c++) {
110                 unsigned int bin = d.get(r, c) / (MAX_ENTRY / BINS);
```

```
111                 b(bin) += 1;
112             }
113         }
114     }
115
116     void print_array(MSA1D::Read& b)
117     {
118         for (unsigned int i=0; i<BINS; ++i) {
119             CkPrintf("%d␣", b.get(i));
120         }
121     }
122 };
123
124 #include "histogram.def.h"
```

Now, in contrast to the typed handle approach, consider the direct access approach shown in listing 5.10. The differences in approach only affect the `Histogram` class, so other portions of the application are omitted. The phase of each array is now implicit in the code, and accesses are not mediated by handle objects. This simplifies and shortens the code, but at the cost of less explicit information about the phase of each array and the lack of an enforcement mechanism for detecting illegal array accesses.

Listing 5.10: The implementation (.cc) file for the Charm++ histogram application with direct array accesses.

```
1  class Histogram: public CBase_Histogram
2  {
3  public:
4      MSA2D data;
5      MSA1D bins;
6
7      Histogram(const MSA2D& data_, const MSA1D& bins_)
8      : data(data_), bins(bins_) {}
9
10     Histogram(CkMigrateMessage* m) {}
11
12     ~Histogram() {}
13
14     // Note: it's important that start is a threaded entry method
15     // so that the blocking MSA calls work as intended.
16     void start()
17     {
```

```
18          if (thisIndex == 0) fill_array(data);
19
20          // transition from write mode to read mode
21          data.sync();
22
23          fill_bins(bins, data);
24
25          // transition from accumulate mode to read mode
26          bins.sync();
27
28          // Print the histogram.
29          if (thisIndex == 0) print_array(bins);
30
31          // Contribute to Driver::done to terminate the program.
32          contribute();
33      }
34
35      void fill_array()
36      {
37          // Just let one thread fill the whole data array
38          // with random entries to be histogrammed.
39          //
40          // Note: this is potentially a very inefficient access
41          // pattern, especially if the MSA doesn't fit into
42          // memory, but it can be convenient.
43          for (unsigned int r = 0; r < data.getRows(); r++) {
44              for (unsigned int c = 0; c < data.getCols(); c++) {
45                  data.set(r, c) = random() % MAX_ENTRY;
46              }
47          }
48      }
49
50      void fill_bins()
51      {
52          // Determine the range of the data array that this
53          // worker should read from.
54          unsigned int range = ROWS / WORKERS;
55          unsigned int min_row = thisIndex * range;
56          unsigned int max_row = (thisIndex + 1) * range;
57
58          // Count the entries that belong to each bin and accumulate
59          // counts into the bins.
60          for (unsigned int r = min_row; r < max_row; r++) {
```

```
61              for (unsigned int c = 0; c < data.getCols(); c++) {
62                  unsigned int bin = data.get(r, c) / (MAX_ENTRY / BINS);
63                  bins(bin) += 1;
64              }
65          }
66      }
67
68      void print_array()
69      {
70          for (unsigned int i=0; i<BINS; ++i) {
71              CkPrintf("%d␣", bins.get(i));
72          }
73      }
74  };
```

The Charj version of this histogram application, given in listing 5.11 is similar to the handle-less approach, but it adds phase names to the synchronization calls (e.g. one might call `syncToAccum` rather than `sync`, but the call is made directly on the MSA in question rather than on a handle object. This adds semantic information about the programmer's intent and improves code readability. Actual detection of MSA access mode violations is done by the compiler. Additionally, array access syntax uses square brackets for consistency with sequential array access syntax, rather than getter/setter functions and overloading of the parentheses operator.

The ability to use MSAs in a message-driven application makes it much simpler to express a variety of interaction patterns that involve unstructured or simply complex sharing of data across processor boundaries, as long as that sharing conforms to a phase structure that can be expressed within MSA. While this is more restrictive than a general-purpose partitioned global address space array package, it provides much greater safety guarantees and offers the possibility of increased scope for runtime optimizations thanks to its rigid phase structure.

Listing 5.11: The core of the Charj version of the histogram application.

```
1   chare Histogram
2   {
3       // Member variables and constructor omitted for brevity
4       public threaded entry void start()
5       {
6           data.syncToWrite();
```

```
7         bins.syncToAccum();
8
9         if (thisIndex == 0) fill_array(data);
10
11        data.syncToRead();
12        fill_bins(bins, data);
13
14        print_array(bins);
15        contribute(null, CkReduction.nop, Driver.done);
16    }
17
18    private void fill_array()
19    {
20        for (unsigned int r = 0; r < data.getRows(); r++) {
21            for (unsigned int c = 0; c < data.getCols(); c++) {
22                data[r, c] = random() % MAX_ENTRY;
23            }
24        }
25    }
26
27    private void fill_bins()
28    {
29        unsigned int range = ROWS / WORKERS;
30        unsigned int min_row = thisIndex * range;
31        unsigned int max_row = (thisIndex + 1) * range;
32
33        for (unsigned int r = min_row; r < max_row; r++) {
34            for (unsigned int c = 0; c < data.getCols(); c++) {
35                unsigned int bin = data[r, c] / (MAX_ENTRY / BINS);
36                bins[bin] += 1;
37            }
38        }
39    }
40
41    private void print_array()
42    {
43        bins.syncToRead();
44        if (thisIndex != 0) return;
45        for (unsigned int i=0; i<BINS; ++i) {
46            CkPrintf("%d␣", bins[i]);
47        }
48    }
49 };
```

5.5 Heterogeneous Computing

The increasing use of floating point accelerator hardware such as general purpose graphical processing units (GPGPUs), field programmable gate arrays (FPGAs), and the Cell processor, and heterogeneous systems that incorporate both traditional multicore processors and accelerators in HPC systems presents a challenge to developers of HPC applications. The high peak performance and energy efficiency associated with accelerators make them attractive targets for compute-intensive HPC codes, but this hardware is widely considered difficult to use effectively, relative to more conventional hardware [56–58].

However, the natural data encapsulation and virtualization provided by the Charm++ runtime system make it well-suited to the effective use of accelerator hardware. This observation led to the development of *accelerated entry methods* [23, 59], which are chare entry methods that the runtime may choose to execute on accelerator hardware (but which may still be executed on a traditional host core. By expressing an application's expensive computational kernels as accelerated entry methods, the programmer allows the runtime system to use available acceleration hardware. This can allow work to be shared between all the different available hardware resources, which increases the scope for dynamically balancing computational load between host and accelerator hardware at runtime.

Accelerated entry methods, as implemented in Charm++, look similar to normal entry methods with a number of syntactic and semantic differences. They are identified with the `accel` keyword and are both defined and declared in the Charm++ interface file, so as to give the translator the requisite information needed to produce both a host implementation and one or more accelerator implementations of the function in question. In addition, accelerated entry methods require some special syntax and have additional restrictions compared to non-accelerated entry methods:

1. In addition to the formal parameters of the method, the programmer must specify which member variables of the parent chare class will be accessed in the body of the function. These are referred to as the *local parameters*. Local parameters are marked as `readOnly`, `writeOnly`, or `readWrite` depending on the needs of the method. Any writeOnly or readWrite local parameters are copied back to the host device at the

end of the method's execution if the execution took place on accelerator hardware.

2. Each accelerated entry method has an associated callback function, specified by the programmer at the end of the function body. This entry method is invoked on the host core when execution of the accelerated entry method is complete.

3. Within the body of the accelerated entry method, the use of some language features is restricted. Most notably, other entry methods may not be invoked from the body of an accelerated entry method.

In other respects, accelerated entry methods are the same as any other entry method. In order to demonstrate the use of accelerated entry methods and illustrate their associated syntax, in listing 5.12 we present simple Charm++ code which takes two matrix tiles as input, multiplies them, and adds the result to a third matrix tile stored locally on the `Tile` chare.

In the listing, the local tile is a variable named `C`, and has `M` rows and `N` columns. Line 3 of the listing contains the local parameter list. It indicates that the local variable `C` in this function corresponds to the chare member variable `C`, and that it is both read and written in the method. The body of the method simply performs the matrix multiply. At the close of the method on line 13, the completion callback `calcTile_callback` is given. This callback will be invoked by the runtime system once the method has completed and, if the execution took place on an accelerator, any modified chare member variables have been copied back to the host core.

5.5.1 Accelerated Entry Methods in Charj

Charj presents several opportunities for simplifying the process of developing applications which make use of accelerated entry methods. Because of the lack of compiler support in the Charm++ implementation, the programmer must manually specify a variety of information that is either readily available to or easily computed by the compiler.

For example, consider the specification of local parameters. Any chare member variables used in the body of an accelerated entry method must be declared in the local parameter declaration block, using syntax of the form:

Listing 5.12: An accelerated entry method for multiplying matrix tiles in Charm++.

```
entry [accel] void calcTile
    (int M, int N, int K, float A[M*K], float B[K*N])
    [ readWrite : float C[M*N] <impl_obj->C> ]
{
    for (int row=0; row<M; ++row) {
        for (int col=0; col<M; ++col) {
            float cv = 0;
            for (int elem=0; elem<K; ++elem)
                cv += A[elem+K*row]*B[col+N*elem];
            C[col+N*row] += cv;
        }
    }
} calcTile_callback;
```

```
access_specifier : type local_name <impl_obj->member_name>
```

where `access_specifier` is one of `readOnly`, `readWrite`, or `writeOnly`, `type` is the variable's type, `local_name` is the name used for the variable in the accelerated entry method, and `member_name` is the name given to the variable in its containing class. This specification allows the generation of code to copy class variables into an accelerator's address space and back out again as necessary. The `impl_obj` syntax is clunky, but it simplifies the code generation process undertaken by the Charm++ translator.

However, all of the information provided in the local parameter declaration is also present in the method body. The information is opaque to the Charm++ translator because it does not parse the C++ method body, but in a Charj implementation of accelerated entry methods, we have full access to it. We need only identify all class variables used in the accelerated entry method, and all potential writes to and reads from these variables.

We use an interprocedural dataflow analysis [55] to identify, for each variable, whether it is only written, only read, or potentially both written and read. We consider a class variable *readable* at a point in a function if there is any path from that point along which it is read, and *writable* if there is any path along which it is written. If we denote the set of variables that is readable at the entrance of basic block b as $readable(b)$, and the corresponding set of writable variables as $writable(b)$, then in order to produce a correct local parameter declaration for a function whose prologue is p, we

Listing 5.13: A Charj equivalent to the Charm++ accelerated entry method in listing 5.12.

```
accelerated entry void calcTile(int M, int N, int K,
        Array<float>A, Array<float> B)
{
    for (int row=0; row<M; ++row) {
        for (int col=0; col<M; ++col) {
            float cv = 0;
            for (int elem=0; elem<K; ++elem)
                cv += A[elem+K*row]*B[col+N*elem];
            C[col+N*row] += cv;
        }
    }
} calcTile_callback;
```

must simply compute $readable(p)$ and $writable(p)$:

$$\begin{aligned} \text{readWrite} &= \text{readable(p)} \cap \text{writable(p)} \\ \text{readOnly} &= \text{readable(p)} - \text{readWrite} \\ \text{writeOnly} &= \text{writable(p)} - \text{readWrite} \end{aligned}$$

Because we are simply computing the union of all reads and writes to variables, the variables that are readable in a block are simply the union of those that are readable at the end of the block and those that are read within the block itself, and similarly for writable variables. We apply this condition iteratively until the sets of readable and writable variables in each block remain unchanged. Because Charj functions may include calls to C++ functions or blocks of C++ code that are not analyzable by the compiler, any local parameter which is available to C++ code is assumed to be both written and read.

By automatically computing the readWrite, readOnly, and writeOnly set, we avoid the need for local parameter declarations in Charj. As a result, accelerated entry methods in Charj look very similar to their unaccelerated siblings, except for the use of the `accelerated` keyword and the presence of the final callback, as shown in listing 5.13.

In addition, the removal of local parameter declarations avoids a possible source of bugs in the Charm++ implementation. Although the programmer must specify whether a given local parameter is read only, write only, or read/write, there is no enforcement or verification mechanism to ensure that

then local parameter in question is actually used in the way specified.

If the programmer wrongly declares a variable to be readonly, any writes to that variable will still occur. If the accelerated entry method happens to be executing on accelerator hardware, those writes will be lost, because readonly local parameters are not copied back to the host when execution completes. However, if the method is executed on the host, the writes will persist. Since the runtime makes dynamic decisions about which hardware to execute on at runtime, this non-deterministic bug may be extremely difficult to identify.

If, on the other hand, a variable is marked as read/write or writeonly and is in fact only ever read, the program will work as intended, but suffer from decreased performance due to unnecessary copying of the local parameter in question.

By eliminating the need to mark the access mode of local parameters, or indeed to declare local parameters at all, the Charj version of accelerated entry methods remove a possible source of programmer error while simplifying the process of writing accelerated entry methods and presenting a more familiar and consistent syntax to the programmer.

Aliasing

Generally, the parameters of entry methods are guaranteed not to alias because each resides in a separate buffer packed by the sender. However, local parameters in accelerated entry methods represent an unusual problem because local parameters are only packed and unpacked in the event that the method is executed on an accelerator. Therefore, if two class variables alias one another, different behavior will be observed if the method executes on the host than if it executes on the accelerator.

Consider the case of two arrays, A and B, which both refer to the same region of memory. In an accelerated entry method, all even indices of A are written to, and all odd indices of B are written to. If this method is executed on the host core, at its completion all of the writes will persist. However, if it is executed on an accelerator, A and B will represent two different buffers on the device, and which ever one is copied back to the host last will be the only one to persist.

We do not detect the potential aliasing of class variables in our analysis of

accelerated entry methods. Even if we did, there is currently no mechanism in the runtime code used to execute accelerated entry methods that would allow for correct behavior in the case of aliased local parameters. So, in this respect Charj shares the same shortcoming of the C++ implementation of accelerated entry methods. In practice, the requirement that local parameters do not alias has not caused any difficulties in application development thus far.

5.6 Summary

The Charm++ runtime system is a capable platform that can support a wide variety of programming models built on top of its message-driven foundation. It offers high performance, flexibility, and the possibility for significant runtime optimizations. However, past models implemented on Charm++ have suffered from inelegance. In the case of multiphase shared arrays, this inelegance stemmed from the difficulty of enforcing programming model semantics from within the confines of a C++ library.

In the case of Structured Dagger and Accelerated Entry Methods, the inelegance stemmed from the use of the Charm++ translator as an ad-hoc compiler for syntax added onto C++. Because the analytical power of the Charm++ translator is relatively limited and because the C++ code that still makes up the bulk of the syntax of both Structured Dagger and Accelerated Entry Methods is entirely opaque to the translator, these models could not take full advantage of the features offered by a compiler, nor were they well integrated with C++ code.

By implementing these programming models within Charj, we integrate them more tightly into mixed codebases, provide clearer syntax to the programmer, eliminate the possibility for common errors while gaining the ability to issue warnings or error messages for problematic code, and create the possibility for model-specific optimizations that would not be possible using the hybrid C++ and translator approach.

CHAPTER 6

OPTIMIZATIONS

This chapter discusses parallel-specific optimizations enabled by the Charj compiler. It starts with a discussion of the basic compiler techniques used, then goes on to describe their application in the context of specific problems in Charm-style parallel applications.

Compiler optimization is ostensibly a tool for improving the performance of programs. It can, when executed well and applied in the correct context, take naive code that ignores important performance issues (potentially specific to a particular hardware architecture) and produce efficient binaries. Viewed in this light, compiler optimizations are a performance-improving technology.

However, compiler optimization can also provide value in the opposite direction, by removing the necessity to write sophisticated code that is made more bulky and obscure because of performance considerations. For the sophisticated and performance-sensitive applications that are typical of the HPC world, the potential benefit of improved compiler optimization is typically not improving the performance characteristics of straightforward but poorly performing code. Rather, it is the replacement of baroque and opaque but high-performing code with simpler and more straightforward code that attains the same performance while becoming more maintainable and more accessible to non-experts. Viewed in this light, compiler optimizations are a productivity-improving technology.

HPC applications tend to be highly optimized by their very nature. Although much productive research and huge amounts of development time

have been dedicated to automatic techniques for improving performance, extensive hand-tuning is still the norm, particularly for performance-sensitive computational kernels.

In part, hand optimization is the product of the need to run efficiently on a large variety of hardware, often while supporting a wide collection of different compilers provided by different vendors. In the case of Charm++, the nightly build tests alone include over a dozen compiler configurations created by multiple providers including GNU, IBM, Microsoft, PGI, and Intel [60]. The variety of hardware and compilers that must be supported by the software prevents developers from relying on optimizations that aren't provided by even the least effective supported compiler configuration. In fact, highly tuned (and self-tuning) software such as ATLAS [61] sometimes goes to substantial lengths to prevent the compiler from trying to perform optimizations that might undo their own performance tweaks and degrade performance.

The need for labor-intensive hand optimizations is also driven in some part by the need to support a large variety of hardware. Even considering only the top 10 supercomputers as ranked by [62] as of November 2011, portable high performance codes must work not only on the familiar Intel Xeon and AMD Opteron multicore processors, but also on Fujitsu SPARC64 and IBM PowerXCell, and NVIDIA GPU architectures. Particularly in the case of Cell and GPU accelerator hardware, programmers must write special-purpose architecture-dependent code to fully take advantage of the hardware's potential.

This diversity of hardware to be supported is a substantial challenge not only to HPC application developers, but also to anyone aiming to provide practical improvements to the compiler optimizations used by HPC programs. Even a very effective optimization that would allow the programmer to substantially simplify his or her program will not produce any simplification of code in practice unless it can be applied across the whole range of architectures and software tool-chains that the program supports.

With Charj, our research agenda is focused on producing enabling technology that simplifies the task of producing high performance parallel programs. Therefore, given high complexity of typical HPC code and the relative sophistication of HPC programmers, compiler optimizations in Charj serve primarily as a tool for enhancing programmer productivity by simplifying or

eliminating the need for program elements that may be complex and bug-prone, repetitive and time-consuming, or time consuming to edit and modify. Our goal is not to advance the state of the art in performance-enabling analysis and optimization, but rather to supply "retail-level" optimizations based on well-understood compiler techniques, but to aim these optimizations directly towards what we believe to be a valuable target: the productivity of practicing HPC application developers. By identifying common but labor-intensive programming tasks, particularly tasks that are specific to parallel application development in a message-driven context, we aim to provide substantial value without the need for groundbreaking analytic techniques.

6.1 Loop Optimizations for MSA

The multiphase shared array programming model (as described in section 5.4) is carefully constructed to allow for a minimum of communication within any given array phase. Minimizing intra-phase communication is important to achieve high performance and prevent thread blocking on accesses.

One of the most common causes of intra-phase communication for an MSA is the reading of array elements that are not locally available. MSA arrays are globally accessible. That is, any thread can access any element of an MSA without regard for its actual location in a distributed memory machine. However, as MSA arrays are distributed data structures, not all array elements are actually present in the thread's local address space.

The actual management of MSA data by the runtime system is somewhat analogous to the way that an operating system might handle virtual memory. The array is decomposed into *pages*, with each page being a member of a chare array that manages the mapping of the array pages onto physical resources. The runtime system maintains a local cache of MSA pages on each processor. Local MSA operations first check the local cache, and only fetch remote pages if the page in question is unavailable locally. The page size of each MSA can be independently set programmatically when the MSA is created. More details on the runtime operations that support the MSA programming model are provided in section 5.4.1.

Given MSA's page caching system, let us now consider its implications for common array access patterns. In listing 6.1, the programmer accesses each

Listing 6.1: A simple loop over a two dimensional $M \times N$ MSA array.

```
1 for (int i=0; i<N; ++i)
2     for (int j=0; j<M; ++j)
3         x += A[j, i];
```

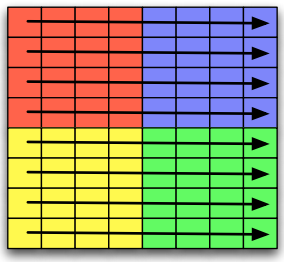


Figure 6.1: The access pattern associated with the code in listing 6.1. Each row of the array is accessed in turn. Arrows from the last element of a row to the first element of the subsequent row are omitted to simplify the diagram. The MSA array has a block-cyclic distribution, and each array page is indicated by its own color.

element of a two dimensional $M \times N$ MSA in row-major fashion. Since the data for this array is distributed across potentially many different address spaces, this series of accesses will involve communication to fetch remote data into the local MSA cache, from which is can be read.

However, since the array access pattern is oblivious to the page structure of the array, it potentially incurs much more communication overhead than is necessary. Suppose the array is laid out in block cyclic fashion, so that each page is a square subregion of the array. Figure 6.1 illustrates the combination of row-major access pattern and block-cyclic data distribution on the array.

For illustrative purposes, suppose that the local MSA page cache holds only a single page, and that this is the only MSA being used by the application. In that case, we incur two page faults per row accessed: one on the first element of the row, and one on the transition between pages that each row contains. In our illustrated example, with $M = N = 8$ and a block size of 4×4, this corresponds to 16 page faults, four for each of the four pages in the array. The accessing thread may share the same location as one or more of these pages, or it may not share the same location as any of them, so the actual communication volume induced by this access pattern is dependent on the runtime distribution of MSA objects. However, in the worst case we send each page to the accessing thread's cache four times, resulting in a worst

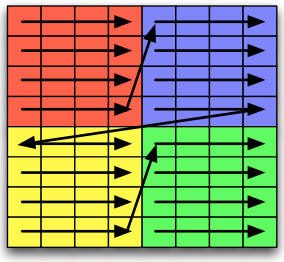


Figure 6.2: The access pattern associated with the code in listing 6.2. Arrows that cross row transitions within a block are omitted to simplify the diagram. The tiled access pattern accesses each element of one MSA page before moving on to the next one.

Listing 6.2: A tiled version of the loop in listing 6.1, with block size $W \times H$. The outer loops iterate over pages, and the inner loops iterate over elements of each page.

```
1 for (int i=0; i<N; i+=H) {
2     for (int j=0; j<M; j+=W) {
3         for (int x=i; x<min(i+H, N); ++x) {
4             for (int y=j; y<min(j+W, M); ++y) {
5                 x += A[j, i];
6             }
7         }
8     }
9 }
```

case communication volume of $4 \times M \times N$ elements.

To reduce this overhead, we must modify the access pattern. By accessing all elements of a page before moving on to the next page, we request each page a maximum of one time and communicate at worst a volume of $M \times N$ elements. Since we never request any element more than once and we can never be guaranteed that any array elements are local to us to begin with, this is the lowest achievable worst case communication volume.

The performance gains that we can achieve on MSA loops is not limited to modifying the access pattern to improve cache performance. We can also explicitly prefetch MSA pages that will be needed in future iterations, as shown in listing 6.3. By invoking prefetch calls one page ahead of the currently accessed page, we effectively overlap the communication involved with transmitting remote pages to the local cache with the computation associated with loop iterations that access the current page.

In addition, by ensuring that the page in question is in the local cache

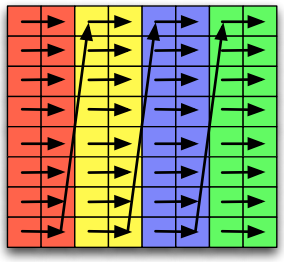


Figure 6.3: If the array is distributed in blocks of columns, a different access pattern would be appropriate. Loop interchange could be used to access entire columns in the inner loop, or the loop order can be preserved while still proceeding block by block. This access pattern results from the same code as shown in listing 6.2 and figure 6.1, but with long, thin blocks rather than square ones. The only values that change are H and W.

before any accesses to the page data, we can improve the performance of array accesses themselves. Because arbitrary accesses to MSA array data may refer to elements that do not exist locally, the default implementation of the MSA data access function involves a conditional check to ensure that the requested element is locally available before it goes on to access the requested element. However, if we can ensure that the data in question is in fact locally available, we can directly access that data via a special accessor function that does not do safety checking to ensure that requested elements are local, thereby improving the performance of each local array access.

This page prefetching technique could form the basis of a more sophisticated adaptive page prefetcher. The amount of prefetching that minimized time to completion depends on a variety of factors that may be known only at runtime, including cache occupancy, network congestion, and available bandwidth. A sufficiently sophisticated prefetching scheme could actively measure these quantities and dynamically adjust the prefetching policy to conform with the current situation. While programmers can and do insert simple prefetching code into their own MSA loops to improve performance, the use of such a system would be highly inconvenient even to a performance-conscious programmer. However, by inserting such code automatically without user intervention, we can make any performance gains associated with this type of system available everywhere without any penalty in code complexity.

Loop tiling is in itself a large and complex area of active research in compil-

Listing 6.3: MSA page prefetching can be used to better overlap communication and computation, improving performance further. We also achieve performance gains by ensuring that all page elements are available locally, then using a raw access function that directly accesses array memory, rather than going through an access layer that first checks for local availability.

```
1 A.prefetch_page(0);
2 for (int i=0; i<N; i+=P) {
3     if (i+P < N)
4         A.prefetch_page(i+P);
5     A.wait_for_page(i);
6     for (int x=i; x<min(i+P, N); ++x) {
7         x += A.raw_access(i);
8     }
9 }
```

ers. Significant work has been done on tiling for complex architectures and complex loop structures [63–65], and even on tiling that explicitly targets parallel architectures and programming models [66, 67].

It is not our goal in this work to extend the frontiers of loop tiling research, nor even to make use of cutting edge performance optimization techniques. Rather, we simply aim to demonstrate the applicability of common compiler techniques to specific performance problems introduced in the context of a multi-model parallel runtime. As such, we have implemented only basic loop tiling for perfectly nested loops without dependencies between iterations. Future efforts to extend this work, for example by introducing polyhedral analysis or improved dependency tracking, would expand the applicability of this optimization to many commonly used forms of loops and thereby broaden its impact on the performance of actual MSA programs.

6.1.1 Possible Library Implementation

Loop tiling is a classic compiler optimization, particularly in the context of cache optimization. As such, it naturally suggested itself as a possible optimization in the context of MSA array accesses. However, it is important to note that this optimization need not be presented to the programmer only in the form of a compiler optimization.

In the case of Charj, we observe the pattern of access to an MSA array, and potentially modify that pattern if it is both safe to do so and we predict

Listing 6.4: Some of the advantages of our MSA loop optimizations could be captured in C++ applications by using an iterator-based approach, which encapsulates tiled array access inside an opaque iterator.

```
1 for (MSA1D<int>::tiled_iterator i = a.begin(); i != a.end(); ++i)
2     x += *i;
3 }
```

that doing so will increase performance, particularly in the form of reduced misses in the MSA page cache. However, one can also pursue an alternate strategy of making the particular array access pattern to be used opaque to the programmer in the text of the program, and making the decision about how to best iterate over array elements within library code that selects an efficient pattern statically.

One natural way to implement this alternative approach is as a C++ iterator, which has the advantage of integrating cleanly with the C++ implementation of MSA. By making the programmer interact with an opaque array handle with an abstract operator that advances to the next element, the writer of the iterator controls the pattern in which the array is accessed. The possible use of such a scheme in application code is demonstrated in listing 6.4.

Such an implementation can perform MSA page tiling and loop prefetching itself, and ensure that array accesses avoid the overhead of availability checking, capturing the same performance benefits of our Charj approach. However, this iterator-based approach is much more limited in the loops that it can potentially support. While we have not implemented any sophisticated dependence analysis in the Charj compiler, we have the potential to support much more general data access patterns than the simple, one element at a time in predetermined order pattern that the iterator-based approach requires. In addition, a compiler-based approach can potentially take advantage of other optimizations, such as code motion, in the context of MSA, while the iterator-based approach has no such possibility. Thus, while the iterator-based approach is potentially quite useful to C++ programmers, its applicability and versatility are potentially much more limited than the compiler-based alternative.

6.2 Optimizing Data Exchange

Empirical studies have sometimes suggested that shared memory programming is more productive than distributed memory [68]. One of the factors that weighs against distributed memory programming in this analysis is the need to pack and unpack application data. Any disagreement between packing code and the corresponding unpacking code can lead to subtle bugs, and the code must be carefully maintained whenever the data being transmitted changes.

In object-oriented programs, the data being transmitted will typically include user-defined types. In most programming models with explicit messaging, the programmer must provide code to handle the packing and unpacking of these types. This support for managing the communication of user-defined types is notable for requiring the programmer to manually specify information that the compiler itself must already know–that is, the types of the variables involved and how they are laid out in memory.

There are many methods by which the code which transmits application data can be created. Perhaps the simplest approach is for the programmer to do the work manually. This largely consists of determining the size of the data to be sent, allocating a buffer of the appropriate size, and then copying the relevant application data into the buffer.

The advantage of this technique is that it is completely customizeable. If a subfield of some user-defined type is needed by a receiver in some portions of an application but not others, the programmer can account for this fact directly. If several variables are known to be contiguous in memory, they can be copied as a block rather than individually.

However, the drawbacks of this approach are obvious. It is a lot of repetitive work to specify all the data that an application transmits in detail, and whenever application data structures change, all the packing and unpacking code has to change with it. It is also error prone, and there is no easy way of verifying that the packing and unpacking is bug-free. While this approach may be feasible, and even high-performance given time and effort, it is extremely poor for productivity.

Alternatively, the programmer may use a library to assist with creating the code. This approach has the advantage that well-designed libraries can significantly ease the process of writing packing and unpacking code while

increasing confidence in that code's correctness. These libraries range from the relatively spartan to full-featured libraries such as Boost.Serialization which include features for cyclic data structures and conditional packing.

However, these libraries typically lack the flexibility to efficiently change the way that an object is packed based on application context. Each field of a type must be either always included or always excluded, leading to inefficiencies. They also require at least some level of intervention by the programmer to integrate their data structures with the library in question.

A large amount of work has been done on data marshalling, both on improving efficiency and on reducing the burden on the programmer. Systems such as Sun RPC [69] provided for marshalling of C structs, using a high-level specification for communication in concert with a stub compiler. Later systems such as CORBA [70] extended this functionality into the object-oriented world. Later work improved the efficiency of generated marshalling code by dynamically choosing between runtime interpretation of data descriptions and compilation [71–73]. However, these systems all require the programmer to explicitly describe the data to be marshalled and do not attempt to determine if any unused data is being transmitted.

More recently there have been several approaches published for providing serialization of C and C++ data structures in MPI applications. For example, C++2MPI [74] and the MPI Preprocessor [75] are both capable of automatically extracting `MPI_Datatype` definitions from C and C++ types. They generate a list of offsets describing the location of all data to be marshalled relative to the base address of the user's data. However, they are limited to marshalling the structure in its entirety and do not handle the case of omitting unneeded data, even in simple cases where the unneeded data does not depend on application context.

AutoMap and AutoLink [76] are also tools that extract MPI datatypes from user code. However, they are limited to C and require the programmer to annotate which fields to pack and which to omit.

Software engineering tools focused on boosting productivity through refactoring have also targeted data marshalling as an area where productivity gains can be had [77]. In [78], Tansey and Telvich describe a graphical tool for generating marshalling code in an MPI context. They allow for multiple versions of the marshalling code to account for the case where different data is needed by the receiver in different application contexts, much as we do

here. However, they rely on the user to manually specify which fields will be packed and which will be omitted in each case, whereas we generate all marshalling code automatically and use compiler analysis to determine which fields to omit.

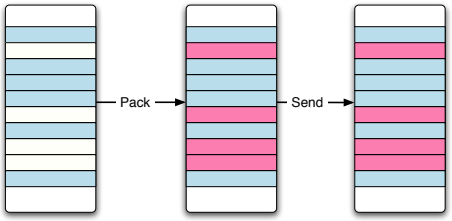
Boost.Serialization takes a library-based approach to providing simple marshalling for C++ datatypes [79]. This library provides largely automatic support for serializing C++ data, but provides no facility for selectively omitting member data depending on context.

Many programming languages explicitly targeted at parallel applications provide automatic marshalling of data or simply present a programming model in which marshalling of user-defined types is not an issue. Generally in programming models where communication is performed via explicit messages marshalling is not entirely automated. This allows the programmer some control over how marshalling takes place. In models where messaging is implicit, the programmer may not even need to consider marshalling. However, in our case we wish to facilitate the productive use of a programming model that does require explicit messaging rather than avoiding the issue altogether.

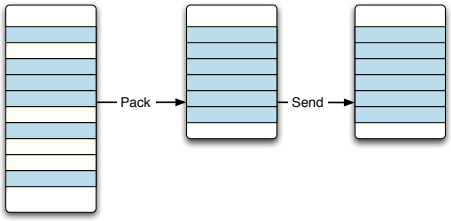
6.2.1 Implementation

We use Charj to address the problem of packing and unpacking application data in a distributed memory environment in a way that minimizes the burden on the programmer while maintaining high performance. We avoid the need for the programmer to manually specify how data structures will be packed and unpacked, and even avoid the need for the programmer to specify which fields of a user-defined type should be packed and which do not need to be sent and can be safely excluded. We do this while producing efficient packing and unpacking code which does not require maintenance when application datatypes or communication patterns are changed.

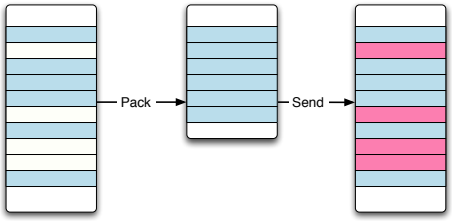
To this end, we use the information available at compile time to generate packing code that guarantees type safety while eliminating the need for manual intervention by the programmer. Because the compiler knows the data layout of each type it can effectively generate packing and unpacking code that does not require updates from the programmer. However, a straight-



(a) The simplest approach is to simply pack the entire data structure regardless of which fields are needed and which are not. This is wasteful of space but maintains encapsulation.



(b) By writing a custom packing routine, the programmer can ensure that no data is unnecessarily transmitted at the cost of breaking encapsulation at the receiving side.



(c) Our technique packs only required fields, but reconstitutes this data on the receiving side as though it was the full object. This maintains encapsulation without wasting bandwidth, but does incur memory overhead on the receiving side.

Figure 6.4: Three approaches to message packing and unpacking. The leftmost box represents a data structure to be sent, and the rectangles inside it represent its fields. The middle box represents the message buffer, and the rightmost box represents the unpacked data at its destination. Fields that are required by the receiving side are colored blue, while wasted memory is colored pink.

forward implementation will still pack data that may not be needed on the receiving side. The programmer can specify which fields to skip, but this requires user intervention and doesn't allow for the possibility that some fields may be needed in one situation but not in another.

One of the benefits of our approach is that it does not require complex or time-intensive compiler analysis. For each remotely invocable method in our application, we wish to produce a function that will pack its arguments, discarding any data which can be proven to be unused. The primary question to be answered is, which variables can be discarded?

Fortunately, there is a simple compiler analysis that answers this question.

Since the function does not interact with its unused fields, the values in those fields are not used in any control flow path that begins at the head of the functions control flow graph. Thus, the function argument fields that are not needed in the body of the function are simply those fields that are not live at the start of the function. Live variable analysis is a well-known and well-studied algorithm [80], so implementation is straightforward. We perform interprocedural analysis where possible, and when code from external libraries is invoked we pessimistically assume that all fields of all arguments to external functions are used.

We treat each user-defined type as a set of elements, with each element corresponding to one field. The output of the live variable analysis is the set of all elements which are live at the function's beginning. Using this set we generate packing code specific to this function which copies each live variable into a buffer, and corresponding unpacking code which reconstitutes the function arguments on the receiving side. To minimize the complexity of our implementation we recreate the full types of all function arguments. This is potentially wasteful of memory, as shown in figure 6.4(c). A better approach would be to transform the receiving function so that instead of expecting the set of arguments specified by the programmer, it instead expects the set of variables that it actually uses. We do not believe that this transformation is difficult, and have left it for future work.

6.2.2 Case Studies

To get a clear idea of how this all works in practice, it is helpful to look at message packing in the context of actual applications. One of the principal advantages of our technique is that it allows the programmer to describe communication in terms natural, high-level objects with semantic meaning rather than simply enumerating the data that will be consumed by the receiving function. However, this benefit cannot be demonstrated on tiny programs like microbenchmarks, because by their nature they are stripped down to the bare essentials needed to perform one task effectively. Thus there are typically no high-level objects that are used in multiple different ways in different contexts, as one would expect in a more realistic application.

To show how our message packing scheme works in an application context

without introducing the full complexities and size of a real production HPC code, we present two case studies taken from the examples provided with the Charm runtime system. These are scaled-down, simplified applications that maintain the structure of more sophisticated scientific codes, but in a smaller and simpler package.

Molecular Dynamics

Charm is best known for NAMD [81], a popular molecular dynamics program in common use at national supercomputing sites. However, NAMD is large and complex, and we do not have the resources that would be required to port NAMD to Charj. However, Charm provides an example molecular dynamics program, named Molecular2D, with similar overall structure to NAMD but with greatly simplified two-dimensional physics. Since this program is provided for pedagogical purposes we might expect it to be written in a way that maximizes clarity at the cost of performance, and in fact this is the case, at least when it comes to message packing.

The primary data structures used in Molecular2D are Particles, which represent the physical objects being modeled, and Patches, which represent a region of space which may contain any number of particles. Listing 6.5 shows the full definition of the Particle type, which mostly consists of information regarding the physical properties of the particle.

The application simulates the motion of these particles over a series of timesteps. In each step, particles within a certain radius exert forces on one another, affecting the position, velocity and acceleration of each. Objects called computes are responsible for managing the interactions between neighboring patches. Each patch sends data regarding its particles to compute objects so that they can determine the effect of those particles on particles belonging to other nearby patches. As the position of a particle changes, it may be migrated from one patch to another.

Listing 6.6 shows the signatures of the functions used by each patch to communicate particle information during each timestep. These are both remotely invoked functions, so their arguments have been marshalled by potentially remote elements. The `updateForces` function is called by a compute which has calculated force contributions to local particles. The function's argument

Listing 6.5: The central particle data structure used by Molecular2D, and its accompanying PUP method.

```
class Particle{
  public:
    int id;
    double mass; // mass
    double pos[2]; // position
    double f[2]; // force
    double a[2]; // acceleration
    double v[2]; // velocity

    void pup(PUP::er &p) {
      p | id;
      p | mass;
      p(pos, 2);
      p(f, 2);
      p(a, 2);
      p(v, 2);
    }
};
```

Listing 6.6: Methods in Molecular2D which receive Particle objects from remote senders. Each takes a list of particles from a remote object which has packed the particle data into a buffer and delivered it to the current patch.

```
class Patch {
    void updateForces(
           vector<Particle> particles);
    void updateParticles(
           vector<Particle> updates);
    // ...
};
```

is a list of particles corresponding to local particles which have forces exerted on them by particles from another patch. The function simply updates the net force on its own particles based on the information it receives from the compute object. The `updateParticles` function migrates particles which have moved outside a patch boundary to the appropriate neighboring patch. This function's argument is a list of formerly remote particles which have moved within the boundaries of the patch during the last timestep.

Semantically, both of these functions operate on a combination of local and remote particle data, so it is natural that they each receive a list of particles as their argument. However, their use of the particle data they receive is quite different. In the case of `updateParticles`, the particles in the list

Listing 6.7: A pup function equivalent to the packing code Charj generates for the `updateForces` method.

```
1 void Particle::pup(PUP::er &p) {
2   p(f, 2);
3 }
```

are migrating to a new patch, and so none of their data can be omitted– each particle will need all of its fields in the next timestep in its new patch. However, this is not the case for `updateForces`. These particles are not migrating, only contributing to the forces exerted on some local particles. Indeed, if we look at the function body in detail, we can see that the only fields of the received particles that used are the forces. The force members represent 16 bytes out of a total of 76 bytes per particle, so nearly 80% of the data transmitted to `updateForces` is pure waste.

In translating this code to Charj, the functions remain mostly unchanged, except that the `pup` function is now unnecessary. However, the actual communication that takes place is much different. During compilation, the `updateParticles` and `updateForces` functions are each analyzed to determine which fields of their arguments are potentially used. In the case of `updateForces` the forces are the only particle components that can possibly be read, so method-specific packing code equivalent to listing 6.7 is generated. In the case of `updateParticles`, the elements of the argument array are added to a data structure belonging to the patch, and from that point on any of their fields could be accessed by Patch methods. Therefore the packing code generated by Charj for this function is equivalent to the full `pup` method of the original application.

N-Body Simulation

The second application we consider is a modified version of the Barnes-Hut N-body algorithm [82] from the well-known SPLASH-2 suite [83]. The modifications are limited to porting the application to use the Charm runtime. The kernel and overall structure of the application remain unchanged.

In this application, a volume of space containing particles is divided into regions using an oct-tree, with each leaf of the tree representing a volume of space that contains an approximately the same number of particles, though

Listing 6.8: A method in the Barnes-Hut application that passes information down the tree. It receives several arguments, each of which is a field of the parent object.

```
1 void recvRootFromParent(uint8_t root_id,
2     double rx, double ry,
3     double rz, double rs);
```

Listing 6.9: A Charj method signature corresponding the the method in listing 6.8.

```
1 void recvRootFromParent(TreePiece parent);
```

the size of these volumes may vary greatly depending on the spatial particle distribution. Then when performing n-body calculations, only particles from nearby volumes must be considered individually, with the contribution of particles from remote volumes only approximated.

The primary communication that takes place in this application is the passing of interaction data up and down the tree. The tree is decomposed into disjoint segments called TreePieces, and data is communicated between pieces via remote invocation of a few methods. Actual transfer of particle data simply uses a vector of particle information in much the same way as the molecular dynamics application described previously. However, information about parent-child relationships within the tree is communicated using other methods of the TreePiece object, such as `recvRootFromParent`.

As shown in listing 6.8, `recvRootFromParent` takes several arguments describing its parent. What is not obvious from the method signature, however, is that each of the arguments comes from a field of the same parent object. However, it is completely impractical to send the entire parent object, because this object contains dozens of fields and a huge amount of data that should not be transmitted.

While the solution adopted by the application of simply splitting out the required data and sending it separately is vastly more efficient, it obscures the origin of the data and the relationship between its arguments. One could preserve this information to some extent by creating a custom type that encapsulates just the information needed for this function, but that approach has high overhead for the programmer, especially in large applications or when an application is being refactored and its arguments change.

Listing 6.9 shows a Charj method signature for the same function. Within the method, uses of `rx` are replaced by `parent.rx`, `ry` by `parent.ry` and so on. This simplifies the method signature, making it easier to see how the function works at a glance. Although each `TreePiece` contains a large number of fields, only the ones used by the receiver are actually transmitted. Thus we get the clarity of the simple code and the efficiency of the more cumbersome, optimized code. In this case the improvement isn't life-changing, but in a larger and more complicated application methods may have dozens of parameters, some subset of which come from a common object and others of which do not. In those cases the simplification may represent a dramatic easing of the burden on the programmer.

6.3 Summary

The creation of highly optimized programs is central to the practice of HPC application development. For performance-critical functions, HPC programmers spend significant time and effort to squeeze as much performance as possible out of their code. Under these circumstances, compiler optimizations tend to function less as tools for increasing application performance, and more as a way a productivity-enhancing tool that allows programmers to use simpler, more straightforward code without sacrificing performance.

Of course, many traditional compiler optimizations apply to HPC domains, often to an even greater extent than less computationally intensive problem domains. In a serial context, performance-improving optimizations for HPC applications have been well studied as part of the broader class of optimizations of interest to the compiler research community.

In this work, it has not been our aim to extend or improve upon the current state of the art for these types of optimizations. Rather, we have attempted to extend the reach of compiler optimizations to our message-driven programming model by applying traditional optimizations in a new context. By applying loop tiling in the context of distributed accesses to multiphase shared arrays and live variable analysis in the context of entry method invocation, we apply simple, well-understood techniques in places that would not be possible without the support of a compiler with explicit knowledge of the parallel programming model in use. These optimizations

demonstrate that there are opportunities within the basic message-driven Charj programming model and the alternate models embedded within it to improve performance through optimization without sacrificing code clarity or brevity. These optimizations may in future form the basis for more a sophisticated and wider-ranging family of optimizations targeted at programs running on the Charm++ runtime system.

CHAPTER 7

WRITING APPLICATIONS IN CHARJ

Ultimately, the goal of improving productivity using Charj cannot be judged outside the context of actual parallel applications. Abstract arguments about clarity and concision and isolated code snippets may be suggestive of benefits, but can never be conclusive on their own. However, given the size and complexity of real, production-ready parallel codes, it is infeasible to create a representative sample of HPC applications in Charj without a massive investment of resources.

Although it is infeasible to produce a suite of full-scale parallel applications in Charj due to the huge amount of developer time and effort that would be required, we can still capture much of the benefit we would gain from such a suite by instead developing stripped-down versions of HPC applications that implement core application functionality while eliminating many of the features that make an application useful for scientists and engineers but which have little bearing on the parallel structure of the application.

In fact, the use of small, self-contained, simplified versions of full applications as a proxy for real, fully-developed applications has gained some popularity in the high performance computing community as way of investigating design trade-offs, algorithm choices, and performance issues [84]. These simplified applications, sometimes referred to as *mini-apps*, take advantage of the fact that even enormous applications with over one million lines of code

often have performance characteristics dominated by a tiny subset of that code, and that of the remainder, these applications can contain a large number of distinct physical models that nevertheless have common performance characteristics [85].

For example, Sandia National Laboratories has developed a suite of mini-apps called Mantevo [86] that aims to provide self-contained open source software that allows for easier analysis of scientific and engineering applications in HPC. It includes mini-apps related to finite element simulation, molecular dynamics, contact detection, and circuit simulation.

7.1 Measuring Productivity

There are many quantities that one can optimize for when writing a parallel application. The most common such quantities in the context of high performance computing are raw performance and scalability. We have argued throughout this dissertation that we must also take into account programmer productivity, and that the techniques outlined here are capable of improving productivity without harming performance and scalability.

However, as we discussed in chapter 1, measuring productivity in a quantifiable way is a difficult problem, particularly when we must consider tradeoffs between the time and effort needed to produce a given code and the performance attained by that code. If we attain a 5% performance improvement and increased scaling at the cost of a 20% increase in programmer time and a 30% increase in lines of code, it is unclear whether that represents a beneficial, productive investment of programmer time or bloating of the codebase and an increase in potential sources of bugs that is not justified by the performance difference. There is no general answer for these questions, and context about the application, programming environment, and programmers in question is necessary to even attempt to provide useful answers.

The problem is even more difficult when we consider comparisons between different programming models. While it may be relatively straightforward to compare the performance of, say, an MPI application and an equivalent Charm++ application, comparing their ease of development and maintenance is difficult and subjective. It is difficult to do a controlled test comparing the development process between two models because of confounding

differences in programmer experience and aptitude and because it is difficult to objectively assess metrics like code maintainability. In addition, a controlled study to assess any productivity benefits conferred by Charj in a realistic setting would require a large investment of programmer time that we have been unable to arrange so far.

Given this limitation, we confine ourselves to attempt to measure productivity indirectly, by producing Charj equivalents of existing parallel applications and comparing them. Because our thesis is that the application of compiler techniques can improve productivity for message driven application development, we make our comparisons to existing message-driven Charm++ applications and do not address the larger issue of comparing message-driven parallel application development to development using other parallel programming models. Since the basic programming model and underlying runtime system is the same in both Charj and Charm++ applications, this allows us to directly compare them and identify areas where Charj features have a concrete impact on either the expressiveness, elegance, and length of the code or on its performance.

To go beyond qualitative comparisons, we must measure concrete attributes of the codes in question. The first and most basic characteristic to measure is source lines of code (SLOC). This is simply a count of the non-empty, non-comment lines of source code in an application. Intuitively, if two programs perform the same tasks using the same techniques, if one is significantly longer than the other then we would prefer the shorter one. However, while lines of code is suggestive of greater productivity it suffers from the fact that a "line" is not an inherently meaningful quantity in a program, and mere formatting conventions can significantly increase or reduce lines of code without altering the application in question. Indeed, taken to an extreme one can compress even very large programs to one enormous line, but we are skeptical of the productivity benefits of this practice in the real world.

One possible alternative metric to lines of code is the cyclomatic number [87], which measures program complexity in terms of the number of nodes and edges in its control flow graph. This metric has the advantage that it is well-defined for any input program independent of its formatting.

However, cyclomatic complexity is in many ways a poor fit for our evaluation of Charj. Because it is primarily concerned with complexity of control flow, the cyclomatic number does not capture the effects of most of Charj's

improvements relative to Charm++. For example, consider the difference between the message-driven and structured dagger (SDAG) implementations of Jacobi relaxation given in section 5.3. The message-driven implementation breaks control flow up across many functions with no clear indication to the programmer how those functions relate to one another, and introduces state variables into the containing class to buffer received messages and keep track of which messages have been received and which are still pending. We argue that this makes it considerably more complex than its SDAG equivalent. We also observe that it is significantly longer than the SDAG implementation. However, the message-driven implementation flattens all local control flow, and therefore has a lower cyclomatic complexity than the SDAG implementation. Simply looking at control flow obscures the advantages that SDAG provides in clarity, convenience, and concision.

Leaving Charj-specific issues aside, the cyclomatic number has been criticized for its weak theoretical justification [88] and has been found to be a worse measure of software complexity than lines of code in numerous studies [89–91]. One alternative, described in [92], is to measure complexity in terms of the number of unique operators and operands and the total number of occurrences of these entities. While this metric, known as *programming effort*, can be effective at measuring the difference in complexity between two very different implementations of an application (i.e. in comparing across different languages or widely differing programming models, as in [93]), a Charj program and its Charm++ equivalent will tend to use the same set of operators on the same operands. In some cases (such as the automated generation of marshalling and unmarshalling functions), Charj will automatically generate code that may require a substantial number of operations in Charm++, but that difference is already captured effectively by lines of code. For this reason, we do not use cyclomatic number or programming effort to compare Charj applications with their Charm++ equivalents.

However, to address some of the shortcomings of SLOC as a complexity metric, we also measure the total number of tokens in each application. This measure is similar to SLOC in that it is mainly concerned with program length, but it improves on SLOC in that it does not depend on the programmer's formatting conventions, and complex lines of code which incorporate inline logic and nested constructs are not favored over a less compact but equally simple alternative formulation. To measure token counts in C++

source files we use an instrumented version of the tokenizer for the Clang C++ compiler [94], and for Charj source and Charm++ interface files we use instrumented versions of `charjc` and `charmxi`, respectively.

7.2 Selecting Applications

We use multiple criteria to select applications for implementation in Charj. First, we only consider algorithms which are well-known and actually used in the HPC community. It is far easier to evaluate the Charj implementation of an algorithm when it can be directly compared to equivalent alternative implementations. By avoiding niche or obscure algorithms, we ensure that our examples are meaningful to a broader audience.

In addition, we want applications that are neither so small that they have little illustrative value and provide little information about the relative difficulty of developing in Charj versus developing directly in Charm++, nor so large that they require inordinate development time and cannot easily be analyzed in full to determine the important points of differentiation between Charj implementations and other implementations. Although this is a rough guideline, we consider applications under 100 lines too small to be useful in our analysis, and applications over 5000 lines too large to be developed effectively while maintaining a large enough collection of Charj applications.

Beyond having a collection of reasonably sized, well known algorithms represented in our suite of applications, we also aim to represent some of the diversity of HPC applications in our sample. Charj is meant to be a general purpose parallel language, and while it might be possible to assemble a collection of different matrix factorization codes that meet all our other criteria, that would give us a very narrow view of Charj's applicability to realistic HPC problems in practice. Rather, we aim to gather a diverse collection of applications that include some of the most common data structures and interaction patterns in HPC. In particular, we wish to represent both matrix-based linear algebra codes and tree-based particle interaction codes.

Finally, when possible we have chosen applications for which high quality Charm++ implementations already exist. Although it is worthwhile to compare Charj applications with their equivalents across widely differing programming models, such a comparison does not necessarily shed any light on

the value provided by the compiler relative to using the underlying runtime system without any compiler support. Rather, it points toward differences between different parallel programming models altogether. While such a comparison is valuable in a discussion about the relative merits of message-driven programming versus message passing versus partitioned global address spaces, for example, it does not directly answer our questions about the benefits that compiler support can add to an existing runtime system. Therefore we prefer to compare our Charj implementations with equivalent Charm++ applications. In particular, we use some submissions from the winning Charm++ entry to the 2011 HPC Class 2 Challenge [95]. These applications are intended as showcases of both high performance and elegant, concise code, and therefore represent a high bar to compare Charj applications against. By selecting pre-existing codes that have been carefully written to exemplify the best combination of elegance and speed possible in Charm++, we ensure that we are comparing ourselves to strong competition in both performance and code size.

With these criteria in mind, in this chapter we present four Charj applications and compare them to their Charm++ equivalents. The first is a Jacobi relaxation application. In it, a 2D array is distributed across a chare array, with each array element containing a contiguous block of columns of the data array. In each iteration, each chare must exchange boundary information with its neighbors. After the neighbor exchange, a simple 5 point stencil is performed on each array element. For the sake of brevity, in this dissertation we will refer to this application simply as "Jacobi."

The second application simulates molecular dynamics based on the Lennard-Jones potential. The application is decomposed according to both space and force. The 3D simulation space is divided into cells, with a chare object responsible for each cell. In each iteration, all pairs of particles within a cutoff distance interact, exerting a force on one another. Pairs of particles that span cells are handled by a separate set of objects called computes, with one compute for each pair of cells within the cutoff distance.

This application, called LeanMD [96], was developed in Charm++ as a simplified version of the popular NAMD [97] molecular dynamics application. It is commonly used as a Charm++ benchmark application .

The final application in our suite is LU Decomposition. This algorithm factorizes an input matrix into a lower triangular matrix (L) and an upper

triangular matrix (U), and is a crucial step in many fundamental numerical algorithms. Although the ideas underlying LU decomposition were known earlier, the modern formulation of the algorithm is credited to Alan Turing [98]. It is a crucial algorithm in HPC, and underlies LINPACK benchmark [99] used to rank supercomputer performance by TOP500 [62].

These applications exhibit substantial variation across a range of application features that are relevant to the HPC community. Jacobi and LU are both primarily concerned with operations on arrays of primitive types, as is typical for a wide variety of HPC applications whose computational kernels are primarily linear algebra. In contrast, LeanMD has much greater reliance on user-defined types. The communication structure of Jacobi is extremely simple and regular: each chare exchanges neighbor information with the same two neighbor chares in each iteration. LeanMD has a more complex but still static communication pattern, in which each Cell communicates with the Computes associated with each of its neighbors within the cutoff distance. LU has a communication pattern that varies as the algorithm proceeds, but which can be statically determined and does not depend on the particular input data.

In the following sections, we describe each of the applications in greater detail, and provide details about differences between Charj and C++-based Charm++ implementations, detailing the ways that Charj features described in previous chapters manifest themselves in the course of actual application development.

7.3 Jacobi Relaxation

Our first application implements an iterative algorithm for solving Laplacian differential equations via Jacobi relaxation. This is a simple solver that can be applied to problems such as heat diffusion. The input to the algorithm is a two dimensional matrix whose entries denote a discretized version of the quantity being simulated (for example, the temperature at one point of a surface). In each timestep, every array element is replaced by the average of its value and the value of its four neighboring elements in a 5-point stencil operation. The application maintains two arrays, and in any given step is reading values from one array and writing the resulting averaged values into

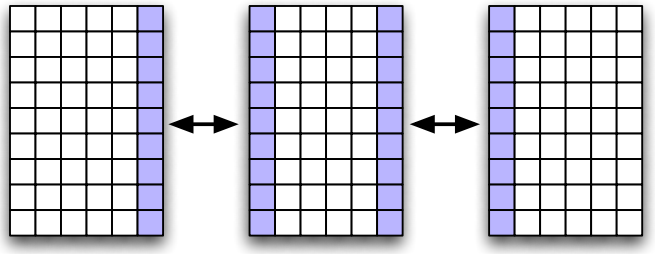


Figure 7.1: In the Jacobi application, the input matrix is decomposed into blocks of columns. In each iteration, the leftmost and rightmost column of each block must be exchanged with neighboring blocks to enable the local update operation to proceed.

the other.

To parallelize the algorithm we perform a two dimensional decomposition of the array into blocks of columns and assign each block to an element of a chare array. Then in each iteration each chunk must communicate with its two neighboring chunks (or one in the case of the leftmost and rightmost chunks) to exchange boundary information before performing its local updates, as depicted in figure 7.3. This pattern of regular communication between fixed neighbors with interspersed local computation is common to many HPC applications, including the MILC quantum chromodynamics simulator [100] and the WRF numerical weather simulator [101, 102].

Our Charj implementation is based on a preexisting Charm++ Jacobi application. We provide abridged code listings that illustrate key elements of the application in both Charj and Charm++ in listings 7.1 and 7.2, respectively. The iterative structure of the application is expressed using Structured Dagger constructs. The original Charm++ implementation does not use an explicit looping construct, probably to avoid the use of class variables as loop indices. Rather, it recursively calls the main loop function until the appropriate number of iterations are complete. The Charj implementation is functionally identical, but uses an SDAG `for` loop directly.

In each iteration, messages containing chunk boundary data are sent to neighboring chunks. Logic for handling the first and last chunk is omitted here and in the stencil function for the sake of brevity. Charj benefits from syntax to concisely describe access to both individual elements and ranges of two dimensional serial arrays, whereas the Charm++ implementation relies on an indexing macro to simplifying array index computations and simple

Application Version	Lines of Code	Tokens
Charj	170	1216
Charm++	397	2807
Percentage Reduction	57.1%	56.7%

Table 7.1: The Charj implementation of the Jacobi application is less than half the size of the equivalent Charm++ application, whether measured in lines of code or in program tokens. This reduction comes partially from the elimination of redundant code in the Charm++ application, and partially from direct support in Charj for two dimensional arrays.

loops over the array to copy boundary elements.

7.3.1 Performance and Productivity

The Charj implementation is less than half the size of the Charm++ implementation, whether measured in lines of code or in program tokens, as shown in table 7.1. The dramatic difference in size is in part due to the small total size of application. Because the application kernels and communication structure are quite brief, the overhead of redundant function and datatype declarations in the Charm++ implementation are exaggerated beyond what one would expect in a larger code. Charj also benefits from syntactic support for two dimensional arrays, both when extracting boundary elements to send to neighboring chunks and when performing the stencil computation itself. Charj also achieves gains relative to Charm in its SDAG implementation and by not needing user-defined message types to transmit data.

We achieve this gain in conciseness of expression while maintaining performance parity with the original Charm++ implementation. We tested the performance of both applications on the 64 nodes of the Taub cluster [103], in which each node has two 2.67 GHz Intel Xeon hex-core processors and 24 GB of RAM, connected by Voltaire QDR Infiniband. We evaluated the applications in a strong scaling scenario, maintaining one million elements per array chunk, and one chunk per physical processor. The results, as shown in figure 7.3.1, do not indicate a performance advantage for either the Charm++ or Charj application.

In principle we expect a small performance penalty in the Charj applica-

Listing 7.1: The time loop, boundary exchange, and stencil computation of the Charj Jacobi application.

```
entry void jacobi() {
    for (int i=0; i<3; ++i) {
        sendStrips();
        overlap {
            when getStripFromLeft(Array<double> s) {
                processStripFromLeft(s);
            }
            when getStripFromRight(Array<double> s) {
                processStripFromRight(s);
            }
        }
        doStencil();
    }
}

void sendStrips() {
    // Send strip left
    if(thisIndex > 0) {
        chunks[thisIndex-1]@getStripFromRight(A[1, 0:myydim]);
    } else {
        // Send dummy message to the last chunk
        chunks[total-1]@getStripFromRight(A[1, 0:myydim);
    }
    // Similarly, send strip right
    // ...
}

void doStencil() {
    resetBoundary(); // clamp boundary region values
    if (thisIndex !=0 && thisIndex != total-1)
        for (int i=1; i<myxdim+1; i++) {
            for (int j=1; j<myydim-1; j++) {
                B[i,j] = (0.2)*(A[i,j] + A[i,j+1] + A[i,j-1] + \
                               A[i+1,j] + A[i-1,j]);
            }
        }
    // similar loops for leftmost and rightmost chunks.
}
```

Listing 7.2: The Charm++ equivalent code for the time loop, boundary exchange, and stencil computation given in listing 7.1.

```
#define indexof(i,j,ydim) ( ((i)*(ydim)) + (j))

entry void singleStep(VoidMsg* msg) {
    atomic "startwork" {
        sendStrips();
    }
    overlap{
        when getStripfromleft(Msg *aMessage){
            atomic {processStripfromleft(aMessage);}
        }
        when getStripfromleftStripfromright(Msg *aMessage){
            atomic {processStripfromright(aMessage);}
        }
    }
    atomic "doWork" {
        doStencil();
        if(iterations < ITER)
            thisProxy[thisIndex].singleStep(new VoidMsg);
    }
}

void sendStrips() {
    // Send strip left
    if (thisIndex > 0) {
        for(int i=0;i<myydim;i++)
            temp[i] = A[indexof(1,i,myydim)];
        chunk_arr[thisIndex-1].getStripfromright(
                new (myydim,0) Msg(myydim,temp));
    } else {
        // Send dummy message to the last chunk
        chunk_arr[total-1].getStripfromright(
                new (myydim,0) Msg(myydim,temp));
    }
    // Similarly, send strip right
    // ...
}

void doStencil() {
    resetBoundary(); // clamp boundary region values
    if (thisIndex !=0 && thisIndex != total-1)
        for (int i=1; i<myxdim+1; i++)
            for (int j=1; j<myydim-1; j++) {
                B[indexof(i,j,myydim)] =
                    (0.2)*(A[indexof(i, j, myydim)] +
                           A[indexof(i, j+1,myydim)] +
                           A[indexof(i, j-1,myydim)] +
                           A[indexof(i+1,j, myydim)] +
                           A[indexof(i-1,j, myydim)]);
            }
    // similar loops for leftmost and rightmost chunks.
}
```

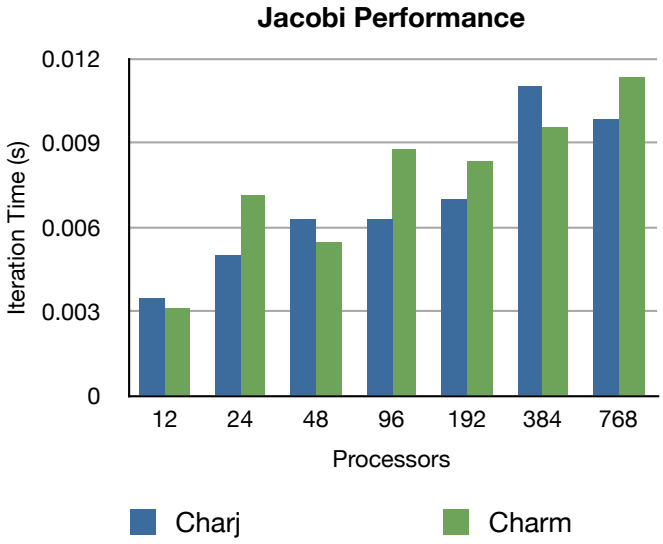


Figure 7.2: Neither Charj nor Charm++ has a clear advantage in performance for the Jacobi application. We measured time per iteration from 12 to 768 processors on the Taub cluster, maintaining a constant one million array elements per processor.

tion, because instead of sending raw array data, we send Charj `Array` types, which contain extra information about array dimensions and layout. In practice, any penalty appears to be dominated by variations in performance from one run to the next.

7.4 Molecular Dynamics

LeanMD is a molecular dynamics simulation that was originally developed as a Charm++ application. It simulates the behavior of atoms based on the Lennard-Jones potential, which describes the interaction between two uncharged molecules or atoms. The computation performed in this code is similar to a simplified version of the short-range non-bonded force calculation in NAMD [104, 105] and resembles the miniMD application in the Mantevo benchmark suite [106] maintained by Sandia National Laboratories.

The force calculation in Lennard-Jones dynamics is done within a cutoff radius, r_c for every atom. For any two particles which are separated by less than the cutoff radius, we explicitly calculate the force that each one

Listing 7.3: The main run loop for the Cell class in the LeanMD application. In each step, the cell sends its local particle positions to Compute objects and receives forces back in return, which it uses to update the particle positions. A reduction after the final iteration is used to validate the simulation.

```
entry void run() {
    if(thisIndex.x==0 && thisIndex.y==0 && thisIndex.z==0) {
        stepTime = CkWallTimer();
    }

    for(int stepCount = 1; stepCount <= finalStepCount; stepCount++) {
        //send current atom positions to my computes
        sendPositions();

        //update properties of atoms using new force values
        when reduceForces(Array<vec3> forces) updateProperties(forces);

        if (thisIndex.x==0 && thisIndex.y==0 &&
                thisIndex.z==0 && stepCount%20==0) {
            CkPrintf("Step␣%d␣Benchmark␣Time␣%lf␣ms/step\n",
                    stepCount, ((CkWallTimer() - stepTime)/20)*1000);
            stepTime = CkWallTimer();
        }
    }
    //everything done, reduction on kinetic energy
    contribute(energy,
            CkReduction.sum_double,
            CkCallback(Main.energySumK, mainProxy));
}
```

exerts on the other. From these forces, we determine particle motion using Newtonian mechanics. To parallelize these computations, we divide the three dimensional simulation space into non-overlapping volumes called cells, with each volume being a rectangular prism. In each timestep, force calculations are performed on every pair of particles that are within the cutoff distance. These calculations are managed by a separate set of chare objects called computes. Each pair of cells within the cutoff distance (including the pair of a cell with itself for computing the forces induced by pairs of particles within a cell) has a compute associated with it that is responsible for computing the interactions between the particles belonging to those cells. Once forces are calculated by the compute objects, the cells perform force integration and update the physical properties of their atoms, including position, velocity, and acceleration.

At the beginning of each time step, every cell sends the positions of its

Listing 7.4: The Charm++ equivalent code for the Cell run loop given in listing 7.3.

```
entry void run() {
    if(thisIndex.x==0 && thisIndex.y==0 && thisIndex.z==0) atomic {
        stepTime = CkWallTimer();
    }

    for(stepCount = 1; stepCount <= finalStepCount; stepCount++) {
        //send current atom positions to my computes
        atomic { sendPositions(); }

        //update properties of atoms using new force values
        when reduceForces(vec3 forces[n], int n) atomic {
            updateProperties(forces, n);
        }

        if (thisIndex.x==0 && thisIndex.y==0 && thisIndex.z==0 &&
                stepCount%20==0) atomic {
            CkPrintf("Step␣%d␣Benchmark␣Time␣%lf␣ms/step\n",
                    stepCount, ((CkWallTimer() - stepTime)/20)*1000);
            stepTime = CkWallTimer();
        }
    }
    //everything done, reduction on kinetic energy
    atomic {
        contribute(2*sizeof(double), energy,
            CkReduction::sum_double,
            CkCallback(CkReductionTarget(Main,energySumK),mainProxy));
    }
};
```

atoms to the computes that need them for force calculations. Every compute receives positions from two cells and calculates the forces. These forces are sent back to the cells which update other properties of the atoms. Every few iterations, atoms are migrated among the cells based on their new positions. SDAG is used to control the flow of operations in each iteration and trigger dependent events. This process is illustrated in the Charj implementation of the Cell class's main loop in listing 7.3, and its Charm++ equivalent in listing 7.3.

7.4.1 Specification and Verification

For a pair of atoms, the force can be calculated based on the distance r between them by

$$\vec{F} = \left(\frac{A}{r^{13}} - \frac{B}{r^{7}} \right) \times \vec{r} \tag{7.1}$$

where A and B are Van der Waals constants. Table 7.2 provides the values for A and B used in the simulation, along with a set of other simulation parameters and the values used in our use of LeanMD.

Parameter	**Values**
A	1.6069×10^{-134}
B	1.0310×10^{-77}
Atoms per cell	150
Cutoff distance, r_c	12 Å
Cell Margin	2 Å
Time step	1 femtosecond

Table 7.2: Simulation parameters for LeanMD. A and B are the Van der Waals constants from equation 7.1.

The application computes the kinetic and potential energy of the simulated system and uses the principle of conservation of energy to verify that the simulation is stable. Users can choose to run the benchmark for as many timesteps as desired, and verification statistics are printed at the end.

Application Version	Lines of Code	Tokens
Charj	570	5190
Charm++	872	7846
Percentage Reduction	34.6%	33.8%

Table 7.3: The Charj implementation of LeanMD is significantly shorter than the Charm++ implementation, both in terms of lines of code and token count.

7.4.2 Performance and Productivity

LeanMD has been developed to be as concise and clear as possible while maintaining high performance as a part of the winning entry for the HPC Challenge in 2011 [95]. The Charm++ implementation of LeanMD is only 773 lines of code, compared to nearly 3000 lines for the Mantevo miniMD benchmark, which has similar goals to and fewer features than LeanMD. As such, this application already represents a high standard of productivity and performance in its original Charm++ embodiment. In our Charj implementation we aim to maintain this high level of performance while simplifying the code.

The Charj implementation of LeanMD is significantly smaller than the original Charm++ application, both in terms of lines of code and token count, as shown in table 7.3. The relative gains are somewhat smaller than in the Jacobi application, between 30% and 35%. This is partially due to the larger overall size of LeanMD compared to Jacobi. The small overall size of the Jacobi application means that the overhead imposed by duplication of interface information across implementation, header, and interface files has a larger impact on the relative sizes of the two codebases relative to LeanMD, where there is significantly more serial implementation code that is quite similar between the Charm++ and Charj implementations.

However, unlike in the Jacobi application, LeanMD user-defined types have PUP methods that allow them to be migrated between processors. These functions enumerate the object fields to be serialized and deserialized when the object is transferred over a network, as shown in listing 7.5. The Charj version of LeanMD gains somewhat in brevity relative to the Charm++ version because it generates these PUP functions automatically, as described in section 6.2. Additionally, as in the Jacobi application, Charj has some

Listing 7.5: The PUP function for the Compute class in the Charm++ implementation of LeanMD. The lack of need for PUP functions is one factor that contributes to the size advantage of the Charj LeanMD implementation.

```
1 void Compute::pup(PUP::er &p) {
2     CBase_Compute::pup(p);
3     __sdag_pup(p);
4     p | stepCount;
5     p | mcast1;
6     p | mcast2;
7     PUParray(p, energy, 2);
8     p | cellCount;
9 }
```

brevity benefits relative to Charm++ in its structured dagger implementation. The Charm++ and Charj versions of the structured dagger function that governs the Cell object's run loop are shown in listings 7.4 and 7.3, respectively. The Charj version benefits from not needing explicitly specified atomic blocks, as even this brief function contains five such blocks.

We expect some reduction in performance in the Charj implementation relative to the Charm++ reference implementation due to the fact that Charj does not provide syntax supporting stack allocation of objects, leading to unneeded extra memory allocation and deallocation when calculating particle interactions. In practice, however, the performance penalty appears to be small. We were unable to conduct a study of the scaling performance of these applications due to lack of availability of cluster resources, but on a four core Nehalem-based workstation, the Charj implementation averaged 219 milliseconds per timestep on a $3 \times 3 \times 3$ cell volume with 300 particles per cell, while the Charm++ implementation averaged 217 milliseconds, a difference of around 1%.

7.5 LU Decomposition

The LU algorithm decomposes the input 2D matrix into square blocks, and associates each block with a member of a chare array. The block data is broadcast to subsequent blocks in the same row or column, and block-block matrix multiplies are used to update local block values, with block level factorizations also being performed for blocks on the matrix diagonal.

To be numerically stable, LU decomposition requires partial pivoting to permute the input matrix rows [98]. However, to keep the application small enough that we can feasibly write and analyze it, we implement a non-pivoting version. Although our implementation is relatively brief at under 200 lines of code, the requirement for partial pivoting creates a huge increase in complexity and code size. For comparison, the HPL 2.0 [107] implementation of LU decomposition with partial pivoting runs to $11,967$ source lines of code and would require 2.71 person-years to develop according to David Wheeler's 'SLOCCount' tool [108]. Although lack of pivoting leads to some loss of numerical stability, the same number of floating point operations are performed by our non-pivoting application when compared to an LU program that implements pivoting [109]. Our implementations (both for Charm++ and Charj) also omit some runtime optimizations that increase performance via careful object mapping and algorithm-specific scheduling policies [110].

The program uses a two dimensional *chare array* to decompose the input matrix into $b \times b$ square blocks. Each matrix block is stored in one of the chare array elements. The mapping of the chare array elements to processors is flexible. The default Charm++ mapping is a block mapping, but other mappings are also possible.

The main computations performed in a dense LU algorithm are matrix-matrix multiplications that update the values in a block. This update operation is referred to as a trailing update. For block (i, j), the block LU algorithm performs $min\,(i, j)$ trailing updates. The closer a block is to the bottom right corner of the overall matrix, the more computation is performed for it. Other computationally intensive portions of the algorithm involve local single-block LU factorizations to be performed for blocks along the diagonal, and updates along the topmost active row and leftmost active column.

Listing 7.6: The Charj implementation of the chare array that contains the matrix blocks for dense LU decomposition. This includes the overall flow of control for the main algorithm in `factorize`, along with kernels that operate on the local block.

```
1 public chare_array [2d] LUBlock {
2     Array<double, 2> LU = new Array<double, 2>([blockSize, blockSize]);
3     int internalStep;
4
5     public entry LUBlock() {
```

```
6           fillRandom(LU);
7       }
8
9       public int min(int a, int b) {
10          if (a < b) return a; else return b;
11      }
12
13      public entry void factorize() {
14          for (internalStep = 0;
15               internalStep < min(thisIndex.x, thisIndex.y);
16               internalStep++) {
17              when recvL[internalStep](Array<double, 2> mL),
18                   recvU[internalStep](Array<double, 2> mU) {
19                      updateMatrix(mL, mU);
20                   }
21          }
22          if (thisIndex.x < thisIndex.y) {
23              // above diagonal
24              when recvL[internalStep](Array<double, 2> mL) {
25                  computeU(mL);
26                  sendDownwardU();
27              }
28          } else if (thisIndex.x > thisIndex.y) {
29              // below diagonal
30              when recvU[internalStep](Array<double, 2> mU) {
31                  computeL(mU);
32                  sendRightwardL();
33              }
34          } else {
35              // on diagonal
36              decompose();
37              if (thisIndex.x < numBlocks - 1 &&
38                      thisIndex.y < numBlocks - 1) {
39                  sendRightwardL();
40                  sendDownwardU();
41              } else {
42                  // implies global completion
43                  CkPrintf("(%d,%d) complete\n", thisIndex.x, thisIndex.y);
44                  driver@finished();
45              }
46          }
47      }
48
```

```
49      public void fillRandom(Array<double, 2> block) {
50          MatGen rnd = new MatGen(9934835);
51          rnd.skipNDoubles(thisIndex.x * blockSize);
52          rnd.getNRndDoubles(block);
53      }
54
55      public void updateMatrix(Array<double, 2> L, Array<double, 2> U) {
56          cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, blockSize,
57                  blockSize, blockSize, -1.0, L.raw(), blockSize,
58                  U.raw(), blockSize, 1.0, LU.raw(), blockSize);
59      }
60
61      public void computeU(Array<double, 2> L) {
62          cblas_dtrsm(CblasRowMajor, CblasLeft, CblasLower,
63                      CblasNoTrans, CblasUnit, blockSize, blockSize,
64                      1.0, L.raw(), blockSize, LU.raw(), blockSize);
65      }
66
67      public void computeL(Array<double, 2> U) {
68          cblas_dtrsm(CblasRowMajor, CblasRight, CblasUpper,
69                      CblasNoTrans, CblasNonUnit, blockSize, blockSize,
70                      1.0, U.raw(), blockSize, LU.raw(), blockSize);
71      }
72
73      public void decompose() {
74          for (int j = 0; j < blockSize; j++) {
75              for (int i = 0; i <= j; i++) {
76                  double sum = 0.0;
77                  for (int k = 0; k < i; k++)
78                      sum += LU[i, k] * LU[k, j];
79                  LU[i, j] -= sum;
80              }
81              for (int i = j + 1; i < blockSize; i++) {
82                  double sum = 0.0;
83                  for (int k = 0; k < j; k++)
84                      sum += LU[i, k] * LU[k, j];
85                  LU[i, j] -= sum;
86                  LU[i, j] /= LU[j, j];
87              }
88          }
89      }
90
91      public void sendRightwardL() {
```

```
92          LUBlock@[%] row =
93              LUBlock@[%].ckNew(thisArrayID,
94                      thisIndex.x, thisIndex.x, 1,
95                      thisIndex.y + 1, numBlocks - 1, 1);
96          row.recvL(LU);
97      }
98
99      public void sendDownwardU() {
100         LUBlock@[%] col =
101             LUBlock@[%].ckNew(thisArrayID,
102                     thisIndex.x + 1, numBlocks - 1, 1,
103                     thisIndex.y, thisIndex.y, 1);
104         col.recvU(LU);
105     }
106 }
```

In the application, matrix blocks are represented by the `LUBlock` chare array. By far the majority of application code resides in LUBlock methods, with the exceptions being application driver code that handles command-line arguments and launches the computation by invoking the `factorize` method on an array of LUBlocks, and a class called `MatGen` which handles the generation of random data for the matrix to be factored.

The full Charj implementation of `LUBlock` is given in listing 7.6. Its functionality falls into two primary categories: linear algebra operations on the local block, and communication with other blocks. The local operations `computeU` and `computeL` are wrappers around the basic linear algebra system (BLAS) function `cblas_dtrsm`, which performs a triangular solve on the local block. Similarly, `updateMatrix` wraps `cblas_dgemm`, which performs matrix multiplication. This functions are invoked on incoming updates that result from local LU decompositions from other blocks in the same row or column to the right or upward from the current block. The final local linear algebra function is `decompose`, which performs an in-place LU decomposition on the local block. Communication with other blocks to send trailing updates is handled by `sendRightwardL` and `sendDownwardU`, which transmit local block data to all the other LUBlocks to the right or down from the current block, respectively.

The overall flow of control of the algorithm is expressed in the structured dagger function `factorize`. First we receive all expected trailing updates from prior blocks via the `recvL` and `recvU` entry methods, performing a

Application Version	Lines of Code	Tokens
Charj	150	1124
Charm++	170	1545
Percentage Reduction	11.8%	27.2%

Table 7.4: Although the sizes of the Charj and Charm++ LU implementations are closer together in terms of lines of code than any other application we consider, the difference in token count is much greater than the difference in line count.

matrix multiply for each pair of updates received. We then perform one of three local block operations, depending on the location of the block. For blocks above or below the diagonal, a local triangular solve is performed once an update from the current diagonal block has been received. For blocks on the diagonal, a local LU decomposition is performed, followed by the transmission of trailing updates via `sendRightwardL` and `sendDownwardU`.

To factorize an $n \times n$ matrix, approximately $\frac{2n^3}{3}$ floating point operations are required. Assuming the matrix is decomposed into $b \times b$ square blocks, the fraction of the floating point operations spent inside the matrix-matrix multiply operation approaches $1 - \frac{1}{b^2}$ as b increases [109]. Thus for large LU factorizations, almost all floating point operations occur in the context of matrix multiplication. Therefore, a performance of a good LU implementation should approach the performance achieved by the double precision matrix-matrix multiply.

7.5.1 Performance and Productivity

In terms of lines of code, Charj has the smallest advantage over Charm++ for LU decomposition out of all the applications that we consider here. As shown in table 7.4, the Charm++ implementation is only about 12% longer than its Charj equivalent. However, the difference in token count is significantly greater, with Charj using over 27% fewer tokens than Charm++.

For the purposes of direct comparison between the two implementations, we include the Charm++ implementation of the LUBlock class that corresponds to the Charj code in listing 7.6 in listings 7.7, 7.8, and 7.9, which contain the interface file definitions and structured dagger implementation,

header declarations, and implementation code, respectively.

The sources of reduced code size in the Charj implementation are largely similar to those from the Jacobi and LeanMD applications. The use of a natural two-dimensional array datatype provides some increase in simplicity versus the C++ implementation, allowing block accesses in `decompose` that look like `LU[i, j]` rather than `LU[i * blocksize + j]`. The triangular solve and matrix multiply kernels are handled by external libraries in the same way for both implementations.

The lines of code advantage for the Charj implementation is somewhat minimized by the fact that there are very few entry methods in the application, and relatively few methods of any kind. While the Charj implementation benefits from smaller token counts in cases of array accesses and atomic blocks in SDAG functions, these advantages do not translate into fewer lines of code. Nevertheless, the 11% advantage that Charj holds in lines of code is not insignificant, and the larger figure for tokens is more in line with its advantages in other small applications.

Listing 7.7: The Charm++ interface definitions for the `LUBlock` data structure, corresponding to the Charj code in listing 7.6.

```
1  array [2d] LUBlock {
2      entry LUBlock();
3      entry void factorize(){
4          atomic {
5              timer = CkWallTimer();
6          }
7          for (internalStep = 0;
8               internalStep < min(thisIndex.x, thisIndex.y);
9               internalStep++) {
10             when recvL[internalStep](int refa,
11                                      int blockSizea,
12                                      double mL[blockSizea]),
13                  recvU[internalStep](int refb,
14                                      int blockSizeb,
15                                      double mU[blockSizeb]) atomic {
16                 updateMatrix(mL, mU);
17             }
18         }
19         if (thisIndex.x < thisIndex.y) {
20             when recvL [internalStep] (int ref,
21                                        int blockSize,
```

```
                                    double mL[blockSize]) atomic {
                computeU(mL);
                sendDownwardU();
            }
        } else {
            if (thisIndex.x > thisIndex.y) {
                when recvU [internalStep] (int ref,
                                           int blockSize,
                                           double mU[blockSize]) atomic {
                    computeL(mU);
                    sendRightwardL();
                }
            } else {
                atomic {
                    decompose();
                    if (thisIndex.x < numBlocks - 1 &&
                        thisIndex.y < numBlocks - 1) {
                        sendRightwardL();
                        sendDownwardU();
                    } else {
                        CkPrintf("(%d,%d) complete, time = %f\n",
                                thisIndex.x, thisIndex.y,
                                CkWallTimer() - timer);
                        driver.finished();
                    }
                }
            }
        }
    };
    entry void recvL(int ref, int blockSize, double mL[blockSize]);
    entry void recvU(int ref, int blockSize, double mL[blockSize]);
};
```

Listing 7.8: The C++ header declarations for the `LUBlock` data structure, corresponding to the Charj code in listing 7.6.

```
class LUBlock: public CBase_LUBlock {
  LUBlock_SDAG_CODE
public:
  double* LU;
  double timer;
  int internalStep;
  LUBlock();
```

```
8   LUBlock(CkMigrateMessage*) { };
9   void fillRandom(double* block);
10  void updateMatrix(double* L, double* U);
11  void computeU(double* L);
12  void computeL(double* U);
13  void decompose();
14  void sendRightwardL();
15  void sendDownwardU();
16 };
```

Listing 7.9: The C++ implementation for the `LUBlock` data structure, corresponding to the Charj code in listing 7.6.

```
1  LUBlock::LUBlock() {
2    __sdag_init();
3    LU = new double[blockSize * blockSize];
4    fillRandom(LU);
5  }
6
7  void LUBlock::fillRandom(double* block) {
8    MatGen rnd(128988);
9    rnd.skipNDoubles(thisIndex.x * blockSize);
10   rnd.getNRndDoubles(block);
11 }
12
13 void LUBlock::updateMatrix(double* L, double* U) {
14   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
15           blockSize, blockSize, blockSize, -1.0,
16           L, blockSize, U, blockSize, 1.0, LU, blockSize);
17 }
18
19 void LUBlock::computeU(double* L) {
20   cblas_dtrsm(CblasRowMajor, CblasLeft, CblasLower,
21           CblasNoTrans, CblasUnit, blockSize, blockSize,
22           1.0, L, blockSize, LU, blockSize);
23 }
24
25 void LUBlock::computeL(double* U) {
26   cblas_dtrsm(CblasRowMajor, CblasRight, CblasUpper,
27           CblasNoTrans, CblasNonUnit, blockSize, blockSize,
28           1.0, U, blockSize, LU, blockSize);
29 }
30
```

```
31 void LUBlock::decompose() {
32   for (int j = 0; j < blockSize; j++) {
33     for (int i = 0; i <= j; i++) {
34       double sum = 0.0;
35       for (int k = 0; k < i; k++)
36         sum += LU[i * blockSize + k] * LU[k * blockSize + j];
37       LU[i * blockSize + j] -= sum;
38     }
39     for (int i = j + 1; i < blockSize; i++) {
40       double sum = 0.0;
41       for (int k = 0; k < j; k++)
42         sum += LU[i * blockSize + k] * LU[k * blockSize + j];
43       LU[i * blockSize + j] -= sum;
44       LU[i * blockSize + j] /= LU[j * blockSize + j];
45     }
46   }
47 }
48
49 void LUBlock::sendRightwardL() {
50   CProxySection_LUBlock row =
51     CProxySection_LUBlock::ckNew(thisArrayID,
52                                  thisIndex.x, thisIndex.x, 1,
53                                  thisIndex.y + 1, numBlocks - 1, 1);
54   row.recvL(internalStep, blockSize*blockSize, LU);
55 }
56
57 void LUBlock::sendDownwardU() {
58   CProxySection_LUBlock col =
59     CProxySection_LUBlock::ckNew(thisArrayID,
60                                  thisIndex.x + 1, numBlocks - 1, 1,
61                                  thisIndex.y, thisIndex.y, 1);
62   col.recvU(internalStep, blockSize*blockSize, LU);
63 }
```

In the LU application, performance is mostly determined by the performance of the system BLAS implementation that performs the block matrix multiplications and triangular solves and by the communication of trailing update information from diagonal blocks. In the Charj implementation of LU, the Charj `Array` type is largely a thin wrapper around the block array data that provides convenient facilities for packing and unpacking that data. Array data is used in place by the BLAS routines with no additional copy overhead. In addition, the pattern of communication is identical in the Charj

application and in the Charm++ equivalent. As such, we expect very similar performance between the two applications.

Here, as in the case of LeanMD, we are unable to provide scaling results due to lack of availability of cluster hardware, but on a four core Nehalem workstation, the Charj LU application factors a 8192×8192 element double precision matrix, decomposed into blocks 128 elements square, in 103.2 seconds. The Charm++ implementation factors the same matrix in 102.9 seconds. These results are consistent with the conclusion that the Charj implementation has very similar performance characteristics to the Charm++ implementation.

7.6 Summary

In this chapter, we have demonstrated the applicability of Charj to three sample applications: Jacobi relaxation, Lennard-Jones molecular dynamics, and dense LU factorization. For each application, we our Charj implementation with a Charm++ equivalent. The reduced size of the Charj implementation relative to the Charm++ baseline ranged from 11.8% to 57.1% in terms of lines of code, and from 27.2% to 56.7% in terms of total program token count. While these numbers are not a direct measure of programmer productivity, the substantial reduction in program size suggests that Charj programs can be written more quickly and easily than their Charm++ equivalents, particularly in light of the fact that that Charj provides benefits to syntax and static checking that are not captured by a simple accounting of lines of code.

In all of these applications, reduction in code size was achieved without sacrificing performance. Despite the significant reduction in code size, the performance-sensitive parts of these applications are very similar in their Charj and Charm++ incarnations. While Charj programs suffer some overheads due to the use of an `Array` wrapper class around access to raw C++ arrays and due to extra memory allocation and deallocation due to the lack of stack-allocated objects, in these applications those overheads amounted to only a few percentage points.

CHAPTER 8

FUTURE WORK

The work discussed in this dissertation is only a part of a wider research agenda to improve the experience of writing high performance parallel applications through the application of compiler technology. A substantial portion of the effort involved in this work has been devoted to the development of a flexible compiler framework that can support not only the features described here, but a wide range of productivity- and performance-improving features that extend upon this work in different directions. Although the possible scope of such features is very large, we will briefly describe a few of the most promising directions for future work in this area.

In chapter 5 we discussed the embedding of other programming models within Charj. We have already embedded multiple models: structured dagger for cleaning expressing message-driven parallel control flow, multiphase shared arrays for disciplined access to global data, and accelerated entry methods for effective support of hybrid accelerator architectures.

There are, however, other existing models based on the Charm++ runtime system that we have not yet translated to Charj, and still more ideas for new models that could benefit from our existing infrastructure. Charisma [111, 112] is an example of the former: an existing model which allows the programmer to express any application with static data flow with a global view of control. Currently it is difficult to integrate Charisma functions in the context of a Charm++ application because of the global view employed by Charisma. Integrating this model into Charj would allow greater opportunities for close

integration between Charisma code and traditional message-passing code.

In addition to the incorporation of existing models, Charj provides a solid foundation upon which to build future models targeted at specific problem domains or patterns of communication. One example model which is already under development targets divide and conquer algorithms, taking advantage of their shared characteristics to provide more robust support for runtime optimizations.

Furthermore, we can build abstractions that rely on static checking to enforce safety properties and ensure high performance. For example, one could implement a capability-based shared array model in which a chare can own a set of capabilities on some subset of an array (such as the ability to read it) and delegate those capabilities to other chares dynamically. With optimization for shared memory architectures, such data structures could be used to efficiently and safely implement algorithms such as parallel quicksort which involve the delegation of array segments to child workers.

We have also implemented optimizations within Charj, most notably loop tiling and prefetching for multiphase shared arrays and communication optimizations that eliminate unused fields from communicated objects. However, these optimizations only scratch the surface of what is possible, particularly with tight integration into runtime services.

The current MSA loop optimization is relatively simple and unsophisticated, in that it does not attempt to determine an optimal amount of prefetching based on machine and application characteristics. However, one could insert instrumentation code that attempts to determine what level of prefetching will maximize performance depending on current conditions. While the current simple prefetching approximates code that a programmer might feasibly write on their own, this kind of adaptive prefetching would go beyond normal optimization practices and potentially increase application performance beyond typical hand-optimized code without sacrificing clarity or elegance.

Similarly, we may be able to use compile-time knowledge of application communication patterns to replace unoptimized communication with special-purpose, optimized runtime code that supports persistent connections and data streaming, thereby improving performance without obscuring the simple pattern of communication expressed by the programmer. Optimizations could also be applied to accelerated entry methods to control grain size and

statically determine whether or not a given method is a good candidate for execution on accelerator hardware.

We can also apply existing optimizations in the context of advanced runtime features. For example, the communication optimization technique used to reduce message sizes for receivers that use only a subset of sent data could also be applied in the areas of fault tolerance and load balancing. In fault tolerance, object data must be written to some data store from which it can be reconstituted in case of a fault. By identifying non-live data at checkpoint time we can reduce checkpoint size and therefore increase performance without requiring the programmer to enumerate the variables that need to be checkpointed. Similarly, load balancing requires the migration of object data from one address space to another, and by omitting data that can not be used at the destination, we reduce the cost of communication associated with load balancing.

All of these potential extensions build upon the fundamental observation that drives this work: that direct compiler support combined with a rich runtime system offers significant opportunities to improve the practice of writing scalable parallel applications. We have demonstrated a variety of techniques that work toward this goal, and we hope that in the future many more will come to join them.

APPENDIX A

CHARJ LANGUAGE GRAMMAR

We provide here a complete grammar for the Charj language. It is specified in a simplified version of the format used by ANTLR's lexer and parser [29]. In the grammar, words in all capitals, such as 'CLASS,' represent a token that consists of the literal word in question, i.e. 'class.' Other literals, such as the binary and unary operators, are simply indicated by quoted strings containing the literal in question. Rules marked as "fragment" can only be matched as components of another top-level rule, and never on their own.

Listing A.1: The C++ implementation for the `LUBlock` data structure, corresponding to the Charj code in listing 7.6.

```
1
2  charjSource
3      : compilationUnit EOF
4
5  compilationUnit
6      : packageDeclaration?
7          topLevelDeclaration+
8
9  topLevelDeclaration
10     : importDeclaration
11     | readonlyDeclaration
12     | externDeclaration
13     | typeDeclaration
14
15 packageDeclaration
```

```
16      : PACKAGE IDENT (DOT IDENT)* ';'!
17
18  importDeclaration
19      : IMPORT^ qualifiedIdentifier '.*'? ';'!
20
21  readonlyDeclaration
22      : READONLY^ localVariableDeclaration ';'!
23
24  externDeclaration
25      : EXTERN^ qualifiedIdentifier ';'!
26
27  typeDeclaration
28      : classDefinition
29      | templateDeclaration
30      | interfaceDefinition
31      | enumDefinition
32      | chareDefinition
33      | messageDefinition
34
35  templateList
36      : 'class'! IDENT (','! 'class'! IDENT)*
37
38  templateDeclaration
39      : 'template' '<'! templateList '>'! classDefinition
40
41  classDefinition
42      : PUBLIC? CLASS IDENT (EXTENDS type)?
43              ('implements' typeList)? '{'! classScopeDeclaration* '}'!
44
45  chareType
46      : CHARE
47      | GROUP
48      | NODEGROUP
49      | MAINCHARE
50      | CHARE_ARRAY '['! ARRAY_DIMENSION ']'!
51
52  chareDefinition
53      : PUBLIC? chareType IDENT (EXTENDS type)?
54              ('implements' typeList)? '{'!
55              classScopeDeclaration*
56          '}'!
57
58  interfaceDefinition
```

```
59      : 'interface' IDENT (EXTENDS typeList)? '{'!
60              interfaceScopeDeclaration*
61          '}'!
62
63  enumDefinition
64      : ENUM IDENT ('implements' typeList)? '{'!
65              enumConstants ','? ';'! classScopeDeclaration*
66          '}'!
67
68  messageDefinition
69      : MESSAGE IDENT '{'! messageScopeDeclaration* '}'!
70      | MULTICAST_MESSAGE IDENT '{'! messageScopeDeclaration* '}'!
71
72  enumConstants
73      : enumConstant (','! enumConstant)*
74
75  enumConstant
76      : IDENT^ arguments?
77
78  typeList
79      : type (','! type)*
80
81  messageScopeDeclaration
82      : primitiveVariableDeclaration
83          | objectVariableDeclaration
84
85  classScopeDeclaration
86      : methodDeclaration
87          | constructorDeclaration
88          | primitiveVariableDeclaration
89          | objectVariableDeclaration
90
91  methodDeclaration
92          : modifierList? genericTypeParameterList? type
93              IDENT formalParameterList (';'! | block)
94
95  constructorDeclaration
96          : modifierList? genericTypeParameterList? ident=IDENT
97              formalParameterList block
98
99  primitiveVariableDeclaration
100         : modifierList? simpleType classFieldDeclaratorList ';'!
101
```

```
102 objectVariableDeclaration
103        : modifierList? objectType classFieldDeclaratorList ';'!
104
105 interfaceScopeDeclaration
106     : modifierList?
107         (genericTypeParameterList?
108             (type IDENT formalParameterList ';'!)
109         | simpleType interfaceFieldDeclaratorList ';'!
110         | objectType interfaceFieldDeclaratorList ';'!)
111
112 classFieldDeclaratorList
113     : classFieldDeclarator (','! classFieldDeclarator)*
114
115 classFieldDeclarator
116     : variableDeclaratorId ('='! variableInitializer)?
117
118 interfaceFieldDeclaratorList
119     : interfaceFieldDeclarator (','! interfaceFieldDeclarator)*
120
121 interfaceFieldDeclarator
122     : variableDeclaratorId ASSIGNMENT! variableInitializer
123
124 variableDeclaratorId
125     : IDENT^ domainExpression?
126
127 variableInitializer
128     : arrayInitializer
129     | expression
130
131 arrayInitializer
132     : '{'! (variableInitializer
133             (','! variableInitializer)* ','?)? '}'!
134
135 templateArg
136     : genericTypeArgument
137     | literal
138
139 templateArgList
140     : templateArg (','! templateArg)*
141
142 templateInstantiation
143     : '<' templateArgList '>'
144     | '<' templateInstantiation '>'
```

```
145
146 genericTypeParameterList
147     : '<' genericTypeParameter (',' genericTypeParameter)* '>'
148
149 genericTypeParameter
150     : IDENT bound?
151
152 bound
153     : EXTENDS type ('&' type)*
154
155 modifierList
156     : modifier+
157
158 modifier
159     : PUBLIC
160     | PROTECTED
161     | ENTRY
162     | SDAGENTRY
163     | TRACED
164     | ACCELERATED
165     | PRIVATE
166     | ABSTRACT
167     | NATIVE
168     | localModifier
169
170 localModifierList
171     : localModifier+
172
173 localModifier
174     : FINAL
175     | STATIC
176     | VOLATILE
177
178 type
179     : simpleType
180     | objectType
181     | VOID
182
183 constructorType
184     : qualifiedTypeIdent AT domainExpression?
185     | qualifiedTypeIdent domainExpression?
186         | MOD qualifiedTypeIdent AT domainExpression
187     | qualifiedTypeIdent TILDE
```

```
188
189 simpleType
190     : primitiveType domainExpression?
191
192 objectType
193     : qualifiedTypeIdent AT domainExpression?
194     | qualifiedTypeIdent domainExpression?
195         | qualifiedTypeIdent '[' MOD ']' AT
196         | qualifiedTypeIdent '[' TILDE ']' AT
197
198 qualifiedTypeIdent
199     : typeIdent (DOT typeIdent)*
200
201 typeIdent
202     : IDENT^ templateInstantiation?
203
204 primitiveType
205     : BOOLEAN
206     | CHAR
207     | BYTE
208     | SHORT
209     | INT
210     | LONG
211     | FLOAT
212     | DOUBLE
213
214 genericTypeArgument
215     : type
216     | '?'
217
218 qualifiedIdentList
219     : qualifiedIdentifier (','! qualifiedIdentifier)*
220
221 formalParameterList
222     : '('! (formalParameterStandardDecl
223               (','! formalParameterStandardDecl)*
224               (','! formalParameterVarArgDecl)?
225               | formalParameterVarArgDecl) ')'!
226
227 formalParameterStandardDecl
228     : localModifierList? type variableDeclaratorId
229
230 formalParameterVarArgDecl
```

```
231     : localModifierList? type '...' variableDeclaratorId
232
233 qualifiedIdentifier
234     : IDENT (DOT IDENT)*
235
236 block
237     : '{'! blockStatement* '}'!
238     | nonBlockStatement
239
240 blockStatement
241     : localVariableDeclaration ';'!
242     | statement
243
244 localVariableDeclaration
245     : primitiveVarDeclaration
246         | objectVarDeclaration
247
248 primitiveVarDeclaration
249         : localModifierList? simpleType classFieldDeclaratorList
250
251 objectVarDeclaration
252         : localModifierList? objectType classFieldDeclaratorList
253
254 statement
255     : nonBlockStatement
256     | sdagStatement
257     | block
258
259 sdagTrigger
260     : IDENT ('['! expression ']'!)? formalParameterList
261
262 sdagStatement
263     : OVERLAP block
264     | WHEN (sdagTrigger (',' sdagTrigger)*)? block
265
266
267 nonBlockStatement
268     : ASSERT expr1=expression
269         (':'! expr2=expression ';'!
270         |';'!)
271     | IF parenthesizedExpression ifStat=block
272         (ELSE elseStat=block | )
273     | FOR '('!
```

```
274         ( forInit? ';'! expression? ';'! expressionList? ')'! block
275         | localModifierList? type IDENT ':'! expression ')'! block
276         )
277     | WHILE parenthesizedExpression block
278     | DO block WHILE parenthesizedExpression ';'!
279     | SWITCH parenthesizedExpression '{'! switchCaseLabel* '}'!
280     | RETURN expression? ';'!
281     | THROW expression ';'!
282     | BREAK IDENT? ';'!
283     | CONTINUE IDENT? ';'!
284     | IDENT ':'! statement
285     | 'delete' expression ';'!
286     | 'embed' STRING_LITERAL EMBED_BLOCK
287     | expression ';'!
288     | ';'
289
290
291 switchCaseLabel
292     : CASE^ expression ':'! blockStatement*
293     | DEFAULT^ ':'! blockStatement*
294
295 forInit
296     : localVariableDeclaration
297     | expressionList
298
299 parenthesizedExpression
300     : '('! expression ')'!
301
302 rangeItem
303     : DECIMAL_LITERAL
304     | IDENT
305
306 rangeExpression
307     : rangeItem
308     | rangeItem ':'! rangeItem
309     | rangeItem ':'! rangeItem ':'! rangeItem
310
311 rangeList
312     : rangeExpression (','! rangeExpression)*
313
314 domainExpression
315     : '['! rangeList ']'!
316
```

```
317 expressionList
318     : expression (','! expression)*
319
320 expression
321     : assignmentExpression
322
323 assignmentExpression
324     : conditionalExpression
325         ( ( ASSIGNMENT^
326             | '+='^
327             | '-='^
328             | '*='^
329             | '/='^
330             | '&='^
331             | '|='^
332             | '^='^
333             | '%='^
334             | '<<='^
335             | '>>='^
336             | '>>>='^)
337         assignmentExpression)?
338
339 conditionalExpression
340     : logicalOrExpression ('?'^ assignmentExpression
341                            ':'! conditionalExpression)?
342
343 logicalOrExpression
344     : logicalAndExpression ('||'^ logicalAndExpression)*
345
346 logicalAndExpression
347     : inclusiveOrExpression ('&&'^ inclusiveOrExpression)*
348
349 inclusiveOrExpression
350     : exclusiveOrExpression ('|'^ exclusiveOrExpression)*
351
352 exclusiveOrExpression
353     : andExpression ('^'^ andExpression)*
354
355 andExpression
356     : equalityExpression ('&'^ equalityExpression)*
357
358 equalityExpression
359     : instanceOfExpression
```

```
360         ( ( '=='^
361             | '!='^
362             )
363             instanceOfExpression
364         )*
365 
366 instanceOfExpression
367     : relationalExpression ('instanceof'^ type)?
368 
369 relationalExpression
370     : shiftExpression
371         ( ( '<='^
372             | '>='^
373             | '<'^
374             | '>'^
375             )
376             shiftExpression
377         )*
378 
379 shiftExpression
380     : additiveExpression
381         ( ( '>>>'^
382             | '>>'^
383             | '<<'^
384             )
385             additiveExpression
386         )*
387 
388 additiveExpression
389     : multiplicativeExpression
390         ( ( '+'^
391             | '-'^
392             )
393             multiplicativeExpression
394         )*
395 
396 multiplicativeExpression
397     : unaryExpression
398         ( ( '*'^
399             | '/'^
400             | '%'^
401             )
402             unaryExpression
```

```
403         )*
404
405 unaryExpression
406     : '+' unaryExpression
407     | '-' unaryExpression
408     | '++' postfixedExpression
409     | '--' postfixedExpression
410     | unaryExpressionNotPlusMinus
411     ;
412
413 unaryExpressionNotPlusMinus
414     : '!' unaryExpression
415     | '~' unaryExpression
416     | '(' type ')' unaryExpression
417     | postfixedExpression
418     ;
419
420 postfixedExpression
421     : primaryExpression
422         ( '.'
423             ( (templateInstantiation? IDENT) (arguments)?
424             | THIS
425             | SUPER arguments
426             | (SUPER '.' IDENT) (arguments)?
427             )
428         | ('@' templateInstantiation? IDENT arguments)
429         | domainExpression
430         )*
431         ('++' | '--')?
432
433 primaryExpression
434     : parenthesizedExpression
435     | literal
436     | newExpression
437     | qualifiedIdentExpression
438     | domainExpression
439     | templateInstantiation
440         ( SUPER (arguments |IDENT arguments)
441         | IDENT arguments
442         | THIS arguments
443         )
444     | THIS (arguments)?
445     | SUPER arguments
```

```
446     | (SUPER DOT IDENT) (arguments |)
447     | SIZEOF '(' expression ')'
448         -> ^(SIZEOF expression)
449     | SIZEOF '(' type ')'
450         -> ^(SIZEOF type)
451 
452 qualifiedIdentExpression
453     : qualifiedIdentifier
454         ( arguments
455         | '.'
456             ( templateInstantiation
457                 ( SUPER arguments
458                 | SUPER '.' IDENT arguments
459                 | IDENT arguments
460                 )
461             | THIS
462             | SUPER arguments
463             )
464         )?
465 
466 newExpression
467     : NEW ( domainExpression arguments? | constructorType arguments)
468 
469 arguments
470     : '('! expressionList? ')'!
471 
472 literal
473     : HEX_LITERAL
474     | OCTAL_LITERAL
475     | DECIMAL_LITERAL
476     | FLOATING_POINT_LITERAL
477     | CHARACTER_LITERAL
478     | STRING_LITERAL
479     | TRUE
480     | FALSE
481     | NULL
482 
483 HEX_LITERAL : '0' ('x'|'X') HEX_DIGIT+ INTEGER_TYPE_SUFFIX?
484 
485 DECIMAL_LITERAL : ('0' | '1'..'9' '0'..'9'*) INTEGER_TYPE_SUFFIX?
486 
487 OCTAL_LITERAL : '0' ('0'..'7')+ INTEGER_TYPE_SUFFIX?
488 
```

```
489 ARRAY_DIMENSION : ('1'..'6')('d'|'D')
490
491 fragment
492 HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F')
493
494 fragment
495 INTEGER_TYPE_SUFFIX : ('l'|'L')
496
497 FLOATING_POINT_LITERAL
498     : ('0'..'9')+
499         (
500             DOT ('0'..'9')* EXPONENT? FLOAT_TYPE_SUFFIX?
501         | EXPONENT FLOAT_TYPE_SUFFIX?
502         | FLOAT_TYPE_SUFFIX
503         )
504     | DOT ('0'..'9')+ EXPONENT? FLOAT_TYPE_SUFFIX?
505
506 fragment
507 EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+
508
509 fragment
510 FLOAT_TYPE_SUFFIX : ('f'|'F'|'d'|'D')
511
512 CHARACTER_LITERAL
513     : '\'' ( ESCAPE_SEQUENCE | ~('\''|'\\') ) '\''
514
515 STRING_LITERAL
516     : '"' ( ESCAPE_SEQUENCE | ~('\\'|'"') )* '"'
517
518 fragment
519 ESCAPE_SEQUENCE
520     : '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
521     | UNICODE_ESCAPE
522     | OCTAL_ESCAPE
523
524 fragment
525 OCTAL_ESCAPE
526     : '\\' ('0'..'3') ('0'..'7') ('0'..'7')
527     | '\\' ('0'..'7') ('0'..'7')
528     | '\\' ('0'..'7')
529
530 fragment
531 UNICODE_ESCAPE
```

```
532     : '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
533
534 IDENT
535     : CHARJ_ID_START (CHARJ_ID_PART)*
536
537 fragment
538 CHARJ_ID_START
539     : '\u0024'
540     | '\u0041'..'\u005a'
541     | '\u005f'
542     | '\u0061'..'\u007a'
543     | '\u00c0'..'\u00d6'
544     | '\u00d8'..'\u00f6'
545     | '\u00f8'..'\u00ff'
546     | '\u0100'..'\u1fff'
547     | '\u3040'..'\u318f'
548     | '\u3300'..'\u337f'
549     | '\u3400'..'\u3d2d'
550     | '\u4e00'..'\u9fff'
551     | '\uf900'..'\ufaff'
552
553 fragment
554 CHARJ_ID_PART
555     : CHARJ_ID_START
556     | '\u0030'..'\u0039'
557
558 WS : (' '|'\r'|'\t'|'\u000C'|'\n')
559
560 fragment
561 EMBED_BLOCK
562     : '{' ( options {greedy=false;} : EMBED_BLOCK | . )* '}'
563
564 COMMENT
565     : '/*' ( options {greedy=false;} : . )* '*/'
566
567 LINE_COMMENT
568     : ('//'|'#') ~('\n'|'\r')* '\r'? '\n'
```

REFERENCES

[1] M. Snir and D. A. Bader, "A framework for measuring supercomputer productivity," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 417–432, Winter 2004. [Online]. Available: http://hpc.sagepub.com/content/18/4/417.abstract

[2] T. Panas, D. Quinlan, and R. Vuduc, "Tool support for inspecting the code quality of hpc applications," in *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications*, ser. SE-HPC '07. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: http://dx.doi.org/10.1109/SE-HPC.2007.8 pp. 2–.

[3] J. Kepner, "Hpc productivity: An overarching view," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 393–397, Winter 2004. [Online]. Available: http://hpc.sagepub.com/content/18/4/393.abstract

[4] T. Sterling, "Productivity metrics and models for high performance computing," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 433–440, Winter 2004. [Online]. Available: http://hpc.sagepub.com/content/18/4/433.abstract

[5] D. J. Kuck, "Productivity in high performance computing," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 4, pp. 489–504, Nov. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004048541

[6] M. O. McCracken, N. Wolter, and A. Snavely, "Beyond performance tools: Measuring and modeling productivity in hpc," in *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications*, ser. SE-HPC '07. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: http://dx.doi.org/10.1109/SE-HPC.2007.2 pp. 4–.

[7] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, and L. Votta, "Measuring high performance computing productivity," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 4, pp. 459–473, Nov. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004048539

[8] W. Gropp, "Learning from the success of mpi," in *High Performance Computing HiPC 2001*, ser. Lecture Notes in Computer Science, B. Monien, V. Prasanna, and S. Vajapeyam, Eds. Springer Berlin / Heidelberg, 2001, vol. 2228, pp. 81–92, 10.1007/3-540-45307-5_8. [Online]. Available: http://dx.doi.org/10.1007/3-540-45307-5_8

[9] L. Hochstein, F. Shull, and L. B. Reid, "The role of mpi in development time: a case study," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413405 pp. 34:1–34:10.

[10] R. Eigenmann and W. Blume, "An effectiveness study of parallelizing compiler techniques," in *ICPP (2)*, 1991, pp. 17–25.

[11] D. Hisley, G. Agrawal, and L. Pollock, "Evaluating the effectiveness of a parallelizing compiler," in *Languages, Compilers, and Run-Time Systems for Scalable Computers*, ser. Lecture Notes in Computer Science, D. OHallaron, Ed. Springer Berlin / Heidelberg, 1998, vol. 1511, pp. 195–204, 10.1007/3-540-49530-414. [Online]. Available: http://dx.doi.org/10.1007/3-540-49530-414

[12] W. Blume and R. Eigenmann, "Performance analysis of parallelizing compilers on the perfect benchmarks programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 643–656, 1992.

[13] M. L. Van De Vanter, A. Wood, C. Vick, S. Faulk, S. Squires, and L. G. Votta, "Productive petascale computing: requirements, hardware, and software," Mountain View, CA, USA, Tech. Rep., 2009.

[14] E. Loh, M. L. Van De Vanter, and L. G. Votta, "Can software engineering solve the hpcs problem?" in *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, ser. SE-HPCS '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1145319.1145328 pp. 27–31.

[15] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[16] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar, "Concurrent collections," *Sci. Program.*, vol. 18, no. 3-4, pp. 203–217, Aug. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1938482.1938486

[17] S. Gunther, "Multi-dsl applications with ruby," *IEEE Software*, vol. 27, pp. 25–30, 2010.

[18] R. K. Brunner, "Versatile automatic load balancing with migratable objects," TR 00-01, January 2000.

[19] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[20] E. Meneses, G. Bronevetsky, and L. V. Kale, "Dynamic load balance for optimized message logging in fault tolerant hpc applications," in *IEEE International Conference on Cluster Computing (Cluster) 2011*, September 2011.

[21] S. Chakravorty and L. V. Kale, "A fault tolerant protocol for massively parallel machines," in *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[22] O. Sarood and L. V. Kalé, "A 'cool' load balancer for parallel applications," in *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[23] L. V. Kale, D. M. Kunzman, and L. Wesolowski, "Accelerator Support in the Charm++ Parallel Programming Model," in *Scientific Computing with Multicore and Accelerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. CRC Press, Taylor & Francis Group, 2011, pp. 393–412.

[24] I. Dooley, "Intelligent runtime tuning of parallel applications with control points," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2010, http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml.

[25] J. DeSouza and L. V. Kalé, "MSA: Multiphase specifically shared arrays," in *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[26] A. Gursoy and L. Kale, "Dagger: Combining the benefits of synchronous and asynchronous communication styles," Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Tech. Rep., March 1993.

[27] L. V. Kale and M. Bhandarkar, "Structured Dagger: A Coordination Language for Message-Driven Programming," in *Proceedings of Second International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 1123-1124, September 1996, pp. 646–653.

[28] P. Miller, A. Becker, and L. Kal, "Using shared arrays in message-driven parallel programs," *Parallel Computing*, vol. 38, no. 12, pp. 66 – 74, 2012.

[29] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995. [Online]. Available: http://dx.doi.org/10.1002/spe.4380250705

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

[31] T. L. Veldhuizen, "C++ templates are turing complete," Tech. Rep., 2003.

[32] J. Sasitorn and R. Cartwright, "Efficient first-class generics on stock java virtual machines," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1141277.1141656 pp. 1621–1628.

[33] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira, "High performance numerical computing in java: Language and compiler issues," in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '99. London, UK, UK: Springer-Verlag, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=645677.663925 pp. 1–17.

[34] L. M. Garshol, "Bnf and ebnf: What are they and how do they work?" 2012. [Online]. Available: http://www.garshol.priv.no/download/text/bnf.html

[35] T. Parr and K. Fisher, "Ll(*): the foundation of the antlr parser generator," *SIGPLAN Not.*, vol. 46, no. 6, pp. 425–436, June 2011. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993548

[36] T. Parr, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, 1st ed. Pragmatic Bookshelf, 2009.

[37] T. J. Parr, "Enforcing strict model-view separation in template engines," in *Proceedings of the 13th international conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/988672.988703 pp. 224–233.

[38] J. Weirich, "Rakefile format documentation," 2012. [Online]. Available: http://rake.rubyforge.org/files/doc/rakefile_rdoc.html

[39] M. Fowler, "Using the rake build language," 2012. [Online]. Available: http://martinfowler.com/articles/rake.html

[40] T. Sloane, "Experiences with Domain-specific Language Embedding in Scala," in *Domain-Specific Program Development*, J. Lawall and L. Réveillère, Eds., Nashville, États-Unis, 2008. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00350269 p. 7.

[41] J. Launchbury, J. R. Lewis, and B. Cook, "On embedding a microarchitectural design language within haskell," in *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ser. ICFP '99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/317636.317784 pp. 60–69.

[42] O. Shivers, "A universal scripting framework or lambda: The ultimate little language," in *Concurrency and Parallelism, Programming, Networking, and Security*, ser. Lecture Notes in Computer Science, J. Jaffar and R. Yap, Eds. Springer Berlin / Heidelberg, 1996, vol. 1179, pp. 254–265, 10.1007/BFb0027798. [Online]. Available: http://dx.doi.org/10.1007/BFb0027798

[43] M. Bravenboer and E. Visser, "Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1028976.1029007 pp. 365–383.

[44] M. Bravenboer, R. de Groot, and E. Visser, "Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt," in *Generative and Transformational Techniques in Software Engineering*, ser. Lecture Notes in Computer Science, R. Lmmel, J. Saraiva, and J. Visser, Eds. Springer Berlin / Heidelberg, 2006, vol. 4143, pp. 297–311, 10.1007/11877028. [Online]. Available: http://dx.doi.org/10.1007/11877028_10

[45] N. Ramsey, "Embedding an interpreted language using higher-order functions and types," in *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, ser. IVME '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/858570.858571 pp. 6–14.

[46] A. Gursoy and L. Kalé, "Dagger: Combining the Benefits of Synchronous and Asynchronous Communication Styles," in *Proceedings of the 8th International Parallel Processing Symposium*, H. G. Siegel, Ed., Cancun, Mexico, April 1994, pp. 590–596.

[47] A. Gursoy and L. Kale, "Tolerating latency with dagger," in *Proceedings of the Eigth International Symposium on Computer and Information Sciences*, Istanbul, Turkey, November 1993.

[48] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *J. Supercomputing*, no. 10, pp. 197–220, 1996.

[49] W. Kuchera and C. Wallace, "The upc memory model: Problems and prospects," 2004.

[50] J. DeSouza, "Jade: Compiler-supported multi-paradigm processor virtualization-based parallel programming," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.

[51] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, 1990. [Online]. Available: citeseer.nj.nec.com/bennett90munin.html pp. 168–177.

[52] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kale, "Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications," *Engineering with Computers*, vol. 22, no. 3-4, pp. 215–235, September 2006.

[53] P. Wadler, "Linear types can change the world!" in *Programming Concepts and Methods*, M. Broy and C. Jones, Eds., 1990.

[54] R. Helm, I. M. Holland, and D. Gangopadhyay, "Contracts: specifying behavioral compositions in object-oriented systems," *SIGPLAN Not.*, vol. 25, no. 10, 1990.

[55] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986. [Online]. Available: http://www.worldcat.org/isbn/0201100886

[56] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Petascale computing with accelerators," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504212 pp. 241–250.

[57] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Programming the linpack benchmark for roadrunner," *IBM Journal of Research and Development*, vol. 53, no. 5, pp. 9:1 –9:11, sept. 2009.

[58] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –8.

[59] D. M. Kunzman, "Runtime support for object-based message-driven parallel applications on heterogeneous clusters," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2012, (to appear).

[60] G. Zheng, "Charm++ automated-build status," 2012. [Online]. Available: https://charm.cs.illinois.edu/autobuild/cur/

[61] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3 – 35, 2001.

[62] "Top500 supercomputing sites," http://top500.org.

[63] N. Ahmed, N. Mateev, and K. Pingali, "Tiling Imperfectly-nested Loop Nests," p. 60, 2000. [Online]. Available: http://citeseer.ist.psu.edu/ahmed00tiling.html

[64] M. Wolfe, "Iteration Space Tiling for Memory Hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1989. [Online]. Available: http://portal.acm.org/citation.cfm?id=645818.669220 pp. 357–361.

[65] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '89. New York, NY, USA: ACM, 1989. [Online]. Available: http://dx.doi.org/10.1145/76263.76337 pp. 655–664.

[66] J. Xue, *Loop tiling for parallelism.* Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[67] J. Ramanujam and P. Sadayappan, "Tiling Multidimensional Iteration Spaces for Multicomputers," *Journal of Parallel and Distributed Computing*, vol. 16, no. 2, pp. 108–230, 1992. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.4376

[68] L. Hochstein and V. R. Basili, "An empirical study to compare two parallel programming models," in *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1148109.1148127 pp. 114–114.

[69] *Remote Procedure Calls: Protocol Specification*, Sun Microsystems, Inc., Mountain View, Calif., May 1988.

[70] *The Common Object Request Broker: Architecture and Specification (Draft)*, 10 December 1991, revision 1.1.

[71] P. Dietz, T. Weigert, and F. Weil, "Formal techniques for automatically generating marshalling code from high-level specifications," in *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, 1998, pp. 40 –47.

[72] N. Feske, "A case study on the cost and benefit of dynamic rpc marshalling for low-level system components," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 40–48, July 2007.

[73] C. Queinnec, "Marshaling/demarshaling as a compilation/interpretation process," *Parallel Processing Symposium, International*, vol. 0, p. 616, 1999.

[74] R. Hillson and M. Iglewski, "C++2mpi: a software tool for automatically generating mpi datatypes from c++ classes," in *Parallel Computing in Electrical Engineering, 2000. PARELEC 2000. Proceedings. International Conference on*, 2000, pp. 13 –17.

[75] E. Renault and C. Parrot, "Mpi pre-processor: generating mpi derived datatypes from c datatypes automatically," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, 0-0 2006, pp. 7 pp. –256.

[76] D. Goujon, M. Michel, J. Peeters, and J. Devaney, "Automap and autolink tools for communicating complex and dynamic data-structures using mpi," in *Network-Based Parallel Computing Communication, Architecture, and Applications*, ser. Lecture Notes in Computer Science, D. Panda and C. Stunkel, Eds. Springer Berlin / Heidelberg, 1998, vol. 1362, pp. 98–109, 10.1007/BFb0052210. [Online]. Available: http://dx.doi.org/10.1007/BFb0052210

[77] D. D. F. Kjolstad and M. Snir, "Bringing the HPC Programmer's IDE into the 21st Century through Refactoring," in *SPLASH 2010 Workshop on Concurrency for the Application Programmer (CAP'10)*. Association for Computing Machinery (ACM), Oct. 2010.

[78] W. Tansey and E. Tilevich, "Efficient automated marshaling of c++ data structures for mpi applications," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –12.

[79] P. Kambadur, D. Gregor, A. Lumsdaine, and A. Dharurkar, "Modernizing the c++ interface to mpi," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, B. Mohr, J. Traff, J. Worringen, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2006, vol. 4192, pp. 266–274.

[80] L. T. Kou, "On live-dead analysis for global data flow problems," *J. ACM*, vol. 24, pp. 473–483, July 1977. [Online]. Available: http://doi.acm.org/10.1145/322017.322027

[81] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "NAMD: A Portable and Highly Scalable Program for Biomolecular Simulations," Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2009-3034, February 2009.

[82] J. E. Barnes and P. Hut, "A hierarchical O(NlogN) force calculation algorithm," *Nature*, vol. 324, 1986.

[83] J. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared memory," *Computer Architecture News*, vol. 20, no. 1, pp. 5–44, March 1992.

[84] A. Geist and S. Dosanjh, "Iesp exascale challenge: Co-design of architectures and algorithms," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 401–402, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1177/1094342009347766

[85] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. R. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep., 2009.

[86] M. Heroux and R. Barrett, "Mantevo project homepage," 2012. [Online]. Available: https://software.sandia.gov/mantevo/index.html

[87] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308 – 320, dec. 1976.

[88] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30 –36, mar 1988.

[89] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *IEEE Trans. Softw. Eng.*, vol. 5, no. 2, pp. 96–104, Mar. 1979. [Online]. Available: http://dx.doi.org/10.1109/TSE.1979.234165

[90] V. Basili and D. Hutchens, "An empirical study of a syntactic complexity family," *Software Engineering, IEEE Transactions on*, vol. SE-9, no. 6, pp. 664 – 672, nov. 1983.

[91] J. L. Elshoff and M. Marcotty, "On the use of the cyclomatic number to measure program complexity," *SIGPLAN Not.*, vol. 13, no. 12, pp. 29–40, Dec. 1978. [Online]. Available: http://doi.acm.org/10.1145/954587.954590

[92] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[93] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua, "Programming with tiles," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 111–122.

[94] C. Lattner, "clang: a c language family frontend for llvm," 2012. [Online]. Available: http://clang.llvm.org/

[95] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng, "Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 11-49, November 2011.

[96] V. Mehta, "LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines," M.S. thesis, University of Illinois at Urbana-Champaign, 2004.

[97] R. Brunner, J. Phillips, and L.V.Kalé, "Scalable molecular dynamics for large biomolecular systems," in *Proceedings of SuperComputing 2000*, 2000.

[98] D. Poole, *Linear Algebra: A Modern Introduction*. Brooks Cole, 2005. [Online]. Available: http://www.worldcat.org/isbn/0534998453

[99] J. Dongarra, "The Linpack Benchmark: An Explanation," in *Evaluating Supercomputers*, A. Van der Steen, Ed. Chapman and Hall, 1990.

[100] M. Collaboration, "MIMD Lattice Computation (MILC) Collaboration Home Page," http://www.physics.indiana.edu/~sg/milc.html.

[101] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers, "A description of the advanced research wrf version 2," NCAR, Tech. Rep. Technical Note NCAR/TN-468+STR, June 2005.

[102] "The weather research & forecasting model website," http://wrf-model.org.

[103] W. Hoyzenga, "Cse parallel computing resource," 2012. [Online]. Available: http://www.cse.illinois.edu/taub/

[104] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "Overcoming scaling challenges in biomolecular simulations across multiple platforms," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[105] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C.Phillips, and C. Harrison, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[106] M. A. Heroux, D. W. Doerer, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep., September 2009.

[107] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers," http://www.netlib.org/benchmark/hpl/.

[108] D. Wheeler, "Sloccount," 2012. [Online]. Available: http://www.dwheeler.com/sloccount/

[109] G. H. Golub and C. F. Van Loan, *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, October 1996.

[110] I. Dooley, C. Mei, J. Lifflander, and L. Kale, "A study of memory-aware scheduling in message driven parallel programs," in *PPL Technical Reports 2010*, no. 10-05, March 2010.

[111] C. Huang and L. V. Kale, "Charisma: Orchestrating migratable parallel objects," in *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[112] P. Jetley and L. V. Kalé, "Static Macro Data Flow: Compiling Global Control into Local Control," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops 2010*, 2010.