COMPILER SUPPORT FOR PRODUCTIVE MESSAGE-DRIVEN PARALLEL PROGRAMMING

Draft of June 6, 2012 at 15 : 47

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 availalble to developers: either make model available to programmers through a library with hooks for common programming languages, or develop a new language altogether. Despite the flexibility of the language approach and the great number of parallel languages that have been created, the library approach exemplified by MPI 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 the user experience that a programming model presents in the ways in which features are exposed to the programmer and in opportunities for optimization and code simplification, drawing from our experiences developing the Charm++ runtime and the Charj language.

Dedication goes here

ACKNOWLEDGMENTS

Acknowledgments go here

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 METHODOLOGY 4
2.1 Objectives 5
2.2 Infrastructure 9

CHAPTER 3 THE CHARJ LANGUAGE 11
3.1 The Charj Programming Model 11
3.2 Charj Syntax 13
3.3 Comparing Charm Applications with Charj Applications . . . 26
3.4 Example Application 27
3.5 Summary 32

CHAPTER 4 THE CHARJ COMPILER 33
4.1 Software Ecosystem 33
4.2 Compiler Architecture 36
4.3 Summary 43

CHAPTER 5 EMBEDDING DIVERSE PROGRAMMING MODELS 44
5.1 Related Work 45
5.2 Supporting Multiple Programming Models 45
5.3 Structured Dagger 46
5.4 Multiphase Shared Arrays 58
5.5 Heterogeneous Computing 74
5.6 Summary 78

CHAPTER 6 WRITING APPLICATIONS IN CHARJ 80
6.1 Conjugate Gradient 81
6.2 Molecular Dynamics 81
6.3 N-Body Simulation 83

6.4 Jacobi Relaxation 88
6.5 LU Decomposition 88

CHAPTER 7 OPTIMIZATIONS 90
7.1 Optimizing Local MSA Array Accesses 92
7.2 Optimizing Data Exchange 92

CHAPTER 8 FUTURE WORK 103

REFERENCES 104

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.

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.

2.2 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.

Make sure this is adequately substantiated in later chapters

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 [15, 16], semi-automated marshalling and unmarshalling of messages, fault tolerance [17, 18], power management [19], use of accelerator architectures [20], control points [21], and many other features.

In addition, multiple programming models already target the Charm runtime [22–25]. 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 [26], 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.

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.

More general indexed collections of chares can be defined using the `chare_array` keyword, which takes an optional dimension argument that specifies the dimensionality 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

need experiment on this to measure overhead.

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 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

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 }
```

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.

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,

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);
};
```

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 pa-

rameters 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 situation 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 [27] 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.

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

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

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 contributed data of mismatched types.

Charj provides an even more pronounced improvement versus Charm++

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.

```
1  // Interface declarations (.ci)
2  initcall void register_my_reducer(void);
3
4  // Implementation (.cc)
5  CkReductionMsg* reduceMyType(int nMsg, CkReductionMsg** msgs)
6  {
7      MyType* accum = new MyType();
8      for (int i=0; i<nMsg; ++i) {
9          MyType* x;
10         PUP::fromMem p(msgs[i]->getData());
11         p | *x;
12         accum->reduce(x);
13     }
14     return CkReductionMsg::buildNew(sizeof(MyType), accum);
15 }
16
17 CkReduction::reducerType _my_reducer_type;
18 void register_my_reducer(void)
19 {
20     _my_reducer_type =
21       CkReduction::addReducer(reduceMyType);
22 }
```

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); }
}
```

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.

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.

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 [28]. 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 [29], 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 [30].

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.

TODO: expand this text and add examples. Maybe ask Jonathan? talk about subranges and library interfaces, too

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 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

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

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     }
17 }
18
19 public chare Fib {
20     Fib@ parent;
21     boolean root;
22     int n;
23     int partialResult;
24     int pendingChildren;
25     const int threshold = 16;
26
27     private int seq_fib(int n) {
28         if (n < 2) return n;
29         return seq_fib(n-1) + seq_fib(n-2);
```

```
30     }
31 
32     public entry Fib(boolean root_, int n_, Fib@ parent_) {
33         n = n_;
34         root = root_;
35         parent = parent_;
36 
37         if (n <= threshold) {
38             partialResult = seq_fib(n);
39             passUp();
40         } else {
41             Fib@ child1 = new Fib@(false, n-1, thisProxy);
42             Fib@ child2 = new Fib@(false, n-2, thisProxy);
43             partialResult = 0;
44             pendingChildren = 2;
45         }
46     }
47 
48     public entry void gather(int value) {
49         partialResult += value;
50         if (--pendingChildren == 0) passUp();
51     }
52 
53     public void passUp() {
54         if (root) main@done(partialResult);
55         else parent@gather(partialResult);
56         delete this;
57     }
58 }
```

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

```
1  mainmodule pgm {
2      readonly CProxy_Main main;
3 
4      mainchare Main {
5          entry Main();
6          entry void done(int value);
7      };
8 
9      chare Fib {
10         entry fib(bool root_, int n_, CProxy_fib parent_);
11         entry void gather(int value);
12     };
```

```
13 };
```

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

```
1  #include "pgm.decl.h"
2
3  class Main : public CBase_Main
4  {
5      public:
6          int n;
7          Main(CkMigrateMessage *m) {}
8          Main(CkArgMsg *m);
9          void done(int value);
10 };
11
12 class Fib : public CBase_Fib
13 {
14     private:
15         int n;
16         int partialResult;
17         int pendingChildren;
18         bool parent;
19         CProxy_fib parent;
20         int seq_fib(int n);
21     public:
22         Fib(CkMigrateMessage *m) {}
23         Fib(bool root_, int n_, CProxy_fib parent_);
24         void gather(int value);
25         void passUp();
26 };
```

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

```
1  #include "pgm.h"
2  #define THRESHOLD 10
3
4  Main::Main(CkArgMsg* m)
5  {
6      if(m->argc < 2) n = 16;
7      else n = atoi(m->argv[1]);
8      main = thisProxy;
9      CProxy_Fib::ckNew(true, n, thishandle);
10 }
```

```
11
12 void Main::done(int value)
13 {
14     CkPrintf("Fib(%d)␣=␣%d\n", n, value);
15     CkExit();
16 }
17
18 Fib::Fib(bool root_, int n_, CProxy_Fib parent_)
19 {
20     root = root_;
21     n = n_;
22     parent = parent_;
23
24     if (n < THRESHOLD) {
25         result = seqFib(n);
26         passUp();
27     } else {
28         CProxy_Fib::ckNew(false, n-1, thishandle);
29         CProxy_Fib::ckNew(false, n-2, thishandle);
30         partialResult = 0;
31         pendingChildren = 2;
32     }
33 }
34
35 int Fib::seqFib(int n) {
36     if (n < 2) return n;
37     return seqFib(n-1) + seqFib(n-2);
38 }
39
40 void Fib::gather(int value) {
41     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 integrating 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 7, 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.

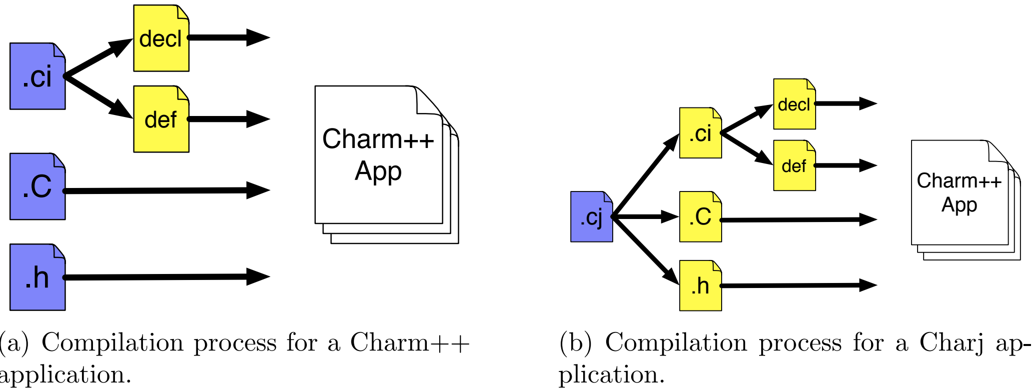
As discussed in chapter 3, Charm++ programs are largely composed of

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 [26], 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.

```
sdagTrigger
    : IDENT ('['! expression ']'!)? formalParameterList
    ;

sdagStatement
    : OVERLAP block
        -> ^(OVERLAP block)
    | WHEN (sdagTrigger (',' sdagTrigger)*)? block
        -> ^(WHEN sdagTrigger* block)
    ;
```

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 [31] 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:

```
^(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 [32]. `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* [33], 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 7.2 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 [34] . 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.

```
1  // Header output template
2  entryMethodDecl_h(classSym, methodSym, modifiers, type, id, params, block) ::=
3  <<
4  <modifiers><type> <id><params>;
5  >>
6
7  // Interface output template
8  entryMethodDecl_ci(classSym, methodSym, modifiers, type, id, params, block) ::=
9  <<
10 <modifiers><type> <id><params>;
11 >>
12
13 // Implementation output template
14 entryMethodDecl_cc(classSym, methodSym, modifiers, type, id, params, block) ::=
15 <<
16 <if(block)>
17 <modifiers><type> <classSym.Name>::<id><params>
18 {
19     <if(methodSym.isTraced)>
20     int _charj_method_trace_timer = CkWallTimer();
21     #endif
22     <endif>
23
24     <block>
25
26     <if(methodSym.isTraced)>
27     traceUserBracketEvent(<methodSym.traceID>,
28             _charj_method_trace_timer, CkWallTimer());
29     <endif>
30 }
31 <endif>
32 >>
```

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.

TODO: discuss and cite examples of each, make case that Charj is novel, good approach.

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) [24, 35, 36] 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.

The equivalent C++ message-driven version of the Jacobi code in listing 5.1 is provided in listing 5.3. The differences between these code listings

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.

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

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.

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

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.

```
entry void jacobi()
{
    i = 0;
    mainLoop();
}

void mainLoop()
{
    leftStripReceived = rightStripReceived = false;
    if (i < N) {
        sendStrips();
    }
}

entry void getStripFromLeft(Strip s)
{
    processStripFromLeft(s);
    leftStripReceived = true;
    checkOverlapCompletion();
}

entry void getStripFromRight(Strip s)
{
    processStripFromRight(s);
    rightStripReceived = true;
    checkOverlapCompletion();
}

void checkOverlapCompletion()
{
    if (leftStripReceived && rightStripReceived) {
        doStencil();
        ++i;
        mainLoop();
    }
}
```

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 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 **??**)* to handle serialization and deserialization of SDAG-specific data structures.

Add section reference on PUP functions once it exists.

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) [22, 25] 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

FIXME: some of this text comes directly from the last MSA paper and needs to be adapted and revised for this context. Limit to factors important to Charj implementations. Some of this knowledge is only needed to understand optimizations, so maybe it needs to be moved/reorganized.

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.

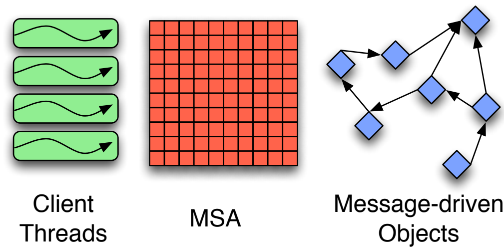
By establishing this discipline, MSA usage is inherently deterministic. However, in exchange for this guarantee, the programmer gives up some of the freedom of a completely general-purpose programming model.

Data Decomposition and Distribution

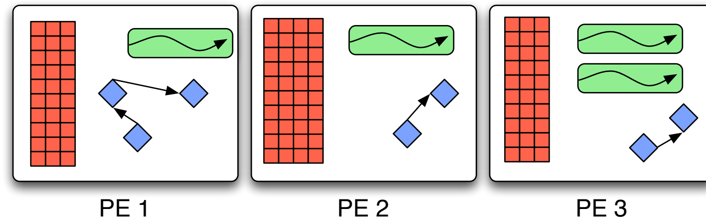
MSA decomposes arrays not into fixed chunks per PE, but rather into *pages* of a common shape. Developers can vary the shape of the pages to suit applications' needs. For example, a 10×10 array could be broken into ten 10×1-shaped pages, or four 5×5 pages, etc. Thus, the library does not couple the number 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



(a) The user view of an MSA application.



(b) One possible mapping of program entities onto PEs

Figure 5.1: 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. FIXME: this graphic needs to be updated/modified for copyright or removed.

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 5.1.

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 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 [37], 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 differ-

ence that MSA's phase structure places much fewer restrictions on communication optimizations than even UPC's most relaxed memory model [38]. 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 7.

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 [39] 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 [40]. 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.

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.

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 [41], uses set union as the accumulation function. The operator's properties guaran-

Listing 5.3: 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.

```
A.syncToWrite();

for (int i = 0; i < N/P; ++i)
    A(tid + i*(P-1)) = f(x, i);

A.syncToRead(); // Done writing A; data can now be read
H.syncToAccum(); // Get ready to increment entries in H

for (int i = 0; i < N/P; ++i) {
    int a = A(i + tid*N/P);
    H(a) += 1;
}
```

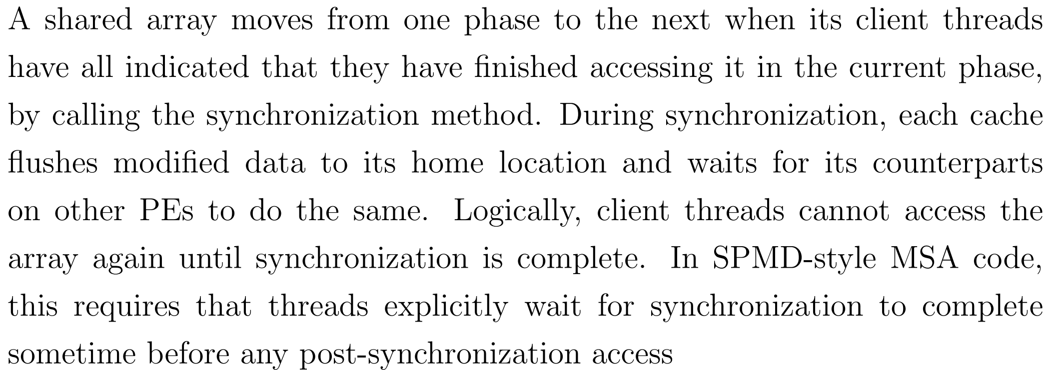
tee that the order in which it's applied does not introduce non-deterministic results.

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.

TODO: why is this a problem for us when it seems to work for, say, std::map?



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 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 [42], 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* approach [43] 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.

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 5.4 and 5.5 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 5.4: The interface (.ci) file for the Charm++ histogram application with typed handles.

```
1 mainmodule histogram
2 {
3     mainchare Driver
4     {
```

```
5       entry void Driver(CkArgMsg*);
6       entry void done(CkReductionMsg*);
7    };
8
9    array [1D] Histogram
10   {
11      entry void Histogram(MSA2D data_, MSA1D bins_);
12      entry [threaded] void start();
13   };
14
15   // Any MSA templates used in the application must be
16   // explicitly instantiated in the interface file.
17   group MSA_CacheGroup<int, DefaultEntry<int>,
18                        MSA_DEFAULT_ENTRIES_PER_PAGE>;
19   array [1D] MSA_PageArray<int, DefaultEntry<int>,
20                            MSA_DEFAULT_ENTRIES_PER_PAGE>;
21 };
```

Listing 5.5: The implementation (.cc) file for the Charm++ histogram application with typed handles.

```
1  #include "msa/msa.h"
2
3  typedef MSA::MSA2D<int, DefaultEntry<int>,
4          MSA_DEFAULT_ENTRIES_PER_PAGE, MSA_ROW_MAJOR> MSA2D;
5  typedef MSA::MSA1D<int, DefaultEntry<int>, MSA_DEFAULT_ENTRIES_PER_PAGE> MSA1D;
6
7  #include "histogram.decl.h"
8
9  const unsigned int ROWS = 2000;
10 const unsigned int COLS = 2000;
11 const unsigned int BINS = 10;
12 const unsigned int MAX_ENTRY = 1000;
13 unsigned int WORKERS = 10;
14
15 class Driver : public CBase_Driver
16 {
17 public:
18    Driver(CkArgMsg* m)
19    {
20       // Usage: histogram [number_of_worker_threads]
21       if (m->argc > 1) WORKERS=atoi(m->argv[1]);
22       delete m;
```

```
23
24          // Actually build the shared arrays: a 2d array to hold arbitrary
25          // data, and a 1d histogram array.
26          MSA2D data(ROWS, COLS, WORKERS);
27          MSA1D bins(BINS, WORKERS);
28
29          // Create worker threads and start them off.
30          workers = CProxy_Histogram::ckNew(data, bins, WORKERS);
31          workers.ckSetReductionClient(
32              new CkCallback(CkIndex_Driver::done(NULL), thisProxy));
33          workers.start();
34      }
35
36      void done(CkReductionMsg* m)
37      {
38          // When the reduction is complete, everything is ready to exit.
39          CkExit();
40      }
41  };
42
43
44  class Histogram: public CBase_Histogram
45  {
46  public:
47      MSA2D data;
48      MSA1D bins;
49
50      Histogram(const MSA2D& data_, const MSA1D& bins_)
51      : data(data_), bins(bins_) {}
52
53      Histogram(CkMigrateMessage* m) {}
54
55      ~Histogram() {}
56
57      // Note: it's important that start is a threaded entry method
58      // so that the blocking MSA calls work as intended.
59      void start()
60      {
61          data.enroll(WORKERS);
62          bins.enroll(WORKERS);
63
64          // Fill the data array with random numbers.
65          MSA2D::Write wd = data.getInitialWrite();
```

```
66          if (thisIndex == 0) fill_array(wd);
67
68          // Fill the histogram bins: read from the data array and
69          // accumulate to the histogram array.
70          MSA2D::Read rd = wd.syncToRead();
71          MSA1D::Accum ab = bins.getInitialAccum();
72          fill_bins(ab, rd);
73
74          // Print the histogram.
75          MSA1D::Read rb = ab.syncToRead();
76          if (thisIndex == 0) print_array(rb);
77
78          // Contribute to Driver::done to terminate the program.
79          contribute();
80      }
81
82      void fill_array(MSA2D::Write& w)
83      {
84          // Just let one thread fill the whole data array
85          // with random entries to be histogrammed.
86          //
87          // Note: this is potentially a very inefficient access
88          // pattern, especially if the MSA doesn't fit into
89          // memory, but it can be convenient.
90          for (unsigned int r = 0; r < data.getRows(); r++) {
91              for (unsigned int c = 0; c < data.getCols(); c++) {
92                  w.set(r, c) = random() % MAX_ENTRY;
93              }
94          }
95      }
96
97      void fill_bins(MSA1D::Accum& b, MSA2D::Read& d)
98      {
99          // Determine the range of the data array that this
100         // worker should read from.
101         unsigned int range = ROWS / WORKERS;
102         unsigned int min_row = thisIndex * range;
103         unsigned int max_row = (thisIndex + 1) * range;
104
105         // Count the entries that belong to each bin and accumulate
106         // counts into the bins.
107         for (unsigned int r = min_row; r < max_row; r++) {
108             for (unsigned int c = 0; c < data.getCols(); c++) {
```

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

Now, in contrast to the typed handle approach, consider the direct access approach shown in listing 5.6. 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 5.6: 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 5.7 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 5.7: 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 [44–46].

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* [20, 47], 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 **??** 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

Listing 5.8: 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;
```

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:

```
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 to identify, for each variable, whether it is only written, only read, or potentially both written and read. Because Charj functions may include calls to C++ functions or blocks of C++ code that are not analyzable by the compiler, any local parameter

Listing 5.9: A Charj equivalent to the Charm++ accelerated entry method in listing 5.8.

```
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;
```

which is reachable from C++ code is assumed to be both written and read. This provides the full set of information needed, and obviates 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 5.9.

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

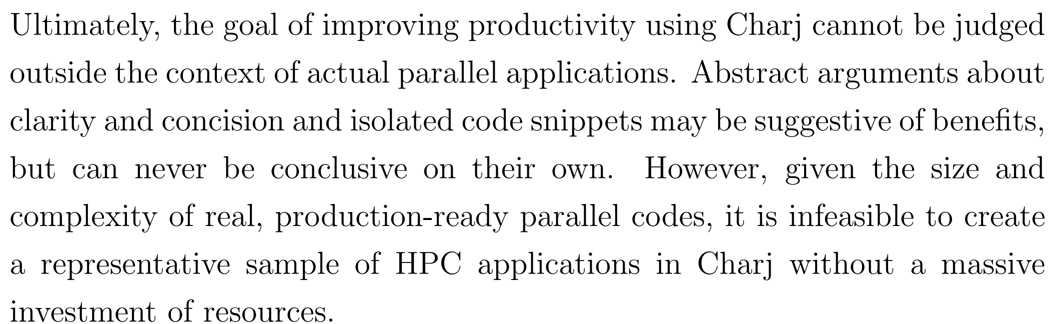
suffered from inelegance. In the case of multiphase shared arrays, this inelegance stemmed from the difficulty of enforcing programming model semantics from within the confines of a C++ library.

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

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 [48]. 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 [49].

For example, Sandia National Laboratories has developed a suite of mini-apps called Mantevo [50] 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.

6.1 Conjugate Gradient

Finite element simulations are among the most important scientific applications in the HPC world. Many of these simulations entail implicit solution of a nonlinear system of equations. As problem size increases, the dominant factor in application performance quickly becomes the implicit solver.

The conjugate gradient kernel represents the bulk of the work involved in many finite element applications. It solves a symmetric system represented by a sparse matrix with no preconditioning. The benchmark generates a 27-point finite difference matrix, and the user defines the size of the matrix sub-blocks that the problem is decomposed into.

The original benchmark was written as an MPI application for the Mantevo project. An equivalent Charm++ version was developed, and our Charj implementation is based on it.

6.2 Molecular Dynamics

TODO: Placeholder text, rewrite

LeanMD is a molecular dynamics simulation program written in Charm++. This benchmark simulates the behavior of atoms based on the Lennard-Jones potential, which is an effective potential that describes the interaction between two uncharged molecules or atoms. The computation performed in this code mimics the short-range non-bonded force calculation in NAMD [51, 52] and resembles the miniMD application in the Mantevo benchmark suite [53]

maintained by Sandia National Laboratories.

The force calculation in Lennard-Jones dynamics is done within a cutoff-radius, r_c for every atom. In LeanMD, the computation is parallelized using a hybrid scheme of spatial and force decomposition. The three-dimensional (3D) simulation space consisting of atoms is divided into cells of dimensions that are equal to the sum of the cutoff distance, r_c and a margin. In each iteration, force calculations are done for all pairs of atoms that are within the cutoff distance. The force calculation for a pair of cells is assigned to a different set of objects called computes. Based on the forces sent by the computes, the cells perform the force integration and update various properties of their atoms – acceleration, velocity and positions.

In the Charm++ implementation of LeanMD, the computation is parallelized using a hybrid scheme of spatial and force decomposition. The three-dimensional (3D) simulation space consisting of atoms is divided into cells of dimensions that are equal to the sum of the cutoff distance, r_c and a margin. In each iteration, force calculations are done for all pairs of atoms that are within the cutoff distance. The force calculation for a pair of cells is assigned to a different set of objects called computes.

At the beginning of each time step, every cell multicasts the positions of its 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. The load balancing framework is invoked periodically after a certain number of iterations to redistribute computes and cells among the processors. In the submitted version, the parallel control flow is described in the `run` functions of each chare in `leanmd.ci`. The reduction for forces computed by computes is in `physics.h`.

6.2.1 Specification and Verification

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

$$\vec{F} = \left(\frac{A}{r^{13}} - \frac{B}{r^{7}} \right) \times \vec{r} \tag{6.1}$$

where A and B are Van der Waals constants and r is the distance between the pair of atoms. Table 6.1 lists a set of parameters and their values used in 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 6.1: Simulation details for LeanMD

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

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 []. The Charm++ implementation of LeanMD is only 773 lines of code, compared to nearly 3000 lines for the Matevo miniMD benchmark, which has similar goals to and fewer features than LeanMD.

6.3 N-Body Simulation

TODO: Placeholder text, rewrite

The N-body problem involves the numerical calculation of the trajectories of N point masses (or charges) moving under the influence of a conservative force field such as that induced by gravity (or electrical charges). In its simplest form, the method models bodies as *particles* of zero extent moving in a collision-less manner. The objective is to calculate the net force incident on every particle at discrete time steps. These forces are then used to update the velocity and position of each particle, leading into the next time step, where the net force on each particle is calculated once more, etc. In general, the force may be long-range in nature (as is the case with gravity), so that interactions between distant particles must also be calculated. Thus,

in order to obtain a good approximation to the actual solution of a system, $O(N^2)$ computations must be performed. Given its quadratic complexity, the amount of work done by this *all-pairs* method makes it infeasible for systems with large N.

Barnes and Hut [54] devised a hierarchical N-body method that performs significantly fewer computations but at the cost of a greater relative error in the computed solution. The method relies on the spatial partitioning of the input system of particles, thereby imposing a tree-structure on it. Particles that are close to each other in space are grouped into closely related nodes of the tree. This allows the approximation of forces on a particle due to a *distant group* of particles through the multipole moments of that group. Note that applying such an approximation to points relatively close to the group will result in gross errors of calculation. In such a case, sub-partitions within the group are tested for proximity to the point. This technique, applied systematically, yields an expected complexity of $O(N lg N)$, making it suitable for large systems of particles.

6.3.1 Implementation

Below we describe the structure of the Barnes-Hut method in greater detail. We also discuss the *distributed* tree data structure used to partition particles and detail its construction.

Space partitioning trees

The Barnes-Hut algorithm relies on the organization of particles into a spatial tree. The leaves of such a tree represent particles, or small groups of particles called *buckets*. Each node of the tree represents a spatial partition enclosing a certain number of particles. Space partitioning trees may be constructed to have different properties and structures. For example the kd-tree attempts to balance the number of particles within each child of a parent node. Here, we focus on the distributed binary *space* partitioning tree as the underlying data structure for the Barnes-Hut algorithm. This data structure has several desirable properties, such as good aspect ratio of partitions (leading to a reduction in the surface area per unit volume, and therefore total communication) and flexibility in deciding the communication grain

size. However, by itself the binary space partitioning tree does not guarantee an even distribution of particles or computational load across partitions. The tree is constructed recursively in the following fashion. Given a node that represents a particular partition of space, if the node has more than a threshold number of particles, it is split along an axis (the axis is chosen in a round-robin fashion) to create two new children of equal size, maintaining the axis-aligned nature of the children.

When constructing the tree in parallel on a distributed memory machine, it is customary to divide the procedure into two distinct phases, namely *domain decomposition* and *tree construction*. In the Charm++ implementation that we discuss here, both phases are managed by a PE-level entity (i.e. a *group*) called the DataManager. Therefore, we will refer to DataManagers (DMs) and PEs interchangeably.

Domain decomposition

The objective of the domain decomposition phase is to assign particles to members of a one-dimensional chare array of *TreePieces*. This step is similar to the sorting of keys in parallel, both in structure and effect. In fact, the iterative *master-worker* structure used here closely resembles that of histogram sort [55]. Each PE (i.e. DM) begins by loading its share of particles from an input file. The PE then performs a local sort on its particles. An iterative distributed histogramming phase follows, in which a master PE obtains the total number of particles in each *active* node. A node is active if the histogramming procedure is currently determining the number of particles in it through a global reduction. The reduction operation is done on arrays of counts contributed by worker PEs. If it is determined that an active node has more than a threshold number of particles, it is removed from the active list and its children are made active. This new list of active nodes is broadcast to the workers, each of which performs a local partitioning of the parents' particles among the new, active children. On the other hand, if a node is determined to be within the user-specified threshold of particles, it is removed from the active list and a corresponding TreePiece is created. At this point, each PE *flushes* the subset of its particles that lie beneath that node to the TreePiece. When there are no active nodes remaining, domain decomposition is complete. The master also broadcasts a list of key ranges to

the workers, thereby informing all PEs of the range of particles held by each TreePiece. This information is needed in order to label the nodes of a local tree with *ownership* data in the tree construction phase (described below).

Tree construction

Upon receiving all the particles intended for it, each TreePiece submits its particles to the DataManager on its PE. This allows all particles on a PE to be agglomerated by the DataManager, resulting in a larger locally accessible tree for all TreePieces on the PE. The DataManager calculates the moments (in our simple implementation these are the total mass and center of mass) of all nodes that are exclusively on its PE. However, in general each PE holds only a subset of the particles in the system, so that the global tree cannot be recreated in its entirety on any PE. Therefore, in order to enable access to remote portions of the tree the DataManager annotates it with ownership information. In effect, the information about the (disjoint) range of particles held by each TreePiece is used to calculate the range of owners of each node. Note that nodes, especially those at shallower levels of the tree, can be shared among many TreePieces since they may enclose particles assigned to a number of TreePieces. In particular, the root is shared by all TreePieces.

Tree traversal

The computation of gravitational forces is preceded by a traversal of the distributed Barnes-Hut tree by each TreePiece. The traversal can be defined recursively, and is at the heart of the $O(N lg N)$ complexity of the algorithm. Given a target particle, on which net force is to be calculated, and a source node (initially the root), the traversal checks whether the distance between the particle and node is large enough to apply the Barnes-Hut approximation. If so, an interaction between the particle and the moments of the node is computed and the node is discarded. If the node is not far enough, the children of the node are considered in turn. In the Charm++ version of the code the DataManager provides TreePieces with seamless access to remote nodes. If the node is a leaf, pairwise interactions between its particles and the target are performed. In our implementation, the cost of traversing the

tree is amortized over several local particles by grouping them into buckets. Of course, by group particles into buckets, we are forcing interactions that needn't be performed, but this extra computation is insignificant when compared to the benefit of reduced memory accesses due to fewer traversals. For each bucket of local particles the global tree is traversed in two parts – a *local* traversal is conducted on that portion of the Barnes-Hut tree that is local to the PE whereas a *remote* traversal operates on the remainder of the distributed Barnes-Hut tree, leading to communication between tree pieces in the form of requests for remote nodes and particles. By assigning greater precedence to remote traversals than local ones, we can use the Charm++ feature of **automatic computation-communication overlap** to accelerate the critical path: in effect, the latency of high-priority remote data communication can be overlapped with low-priority local work.

Requests for remote data are funneled through the DataManager so as to merge requests for the same particles and nodes from local TreePieces. This optimization in itself can substantially reduce the volume of communication. We combine this technique with a software-managed remote data cache to increase the amount of data reuse. Upon receipt of remote data, the DataManager maintains a copy of the data so that it may be reused by other TreePieces that require it.

Advancing particles

Once the net force on each particle has been calculated, we integrate the kinematical equations of motions for it over the duration of a single time step. We use a single-timestepping second-order leapfrog integration technique for this purpose. Note that this translation can only be performed once it is guaranteed that all traversals have completed. This serves as the boundary for an iteration; following it, particles are once again decomposed onto TreePieces, this time based on the new positions of the particles.

6.4 Jacobi Relaxation

6.5 LU Decomposition

TODO: Placeholder text, rewrite

The LU program was written as simply as possible, without any explicit memory-awareness in the parallel program's code. This implementation does not perform pivoting. Hence some numerical stability is lost, but the same number of floating point operations are still performed when compared to an LU program that implements pivoting [56].

The program uses a 2-D *chare array* to decompose the 2-D 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 the program can easily specify other mappings.

The main communication pattern that occurs in an LU matrix factorization is a multicast of a data block from a source block to all subsequent blocks in the same row, and a downward multicast of a data block from its source to all blocks below it in the same column. The Charm++ language natively supports chare array section sends, which are a mechanism for sending a single message to a set of destination chare array elements. The programmer can choose one of many predefined algorithms for each section send [57]. The Charm++ LU implementation can therefore easily represent the pattern of communication that needs to occur. The multicast algorithm that appears to perform well for the cases described below uses a simple processor spanning tree of degree 4.

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.

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 [56]. 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.

CHAPTER 7

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 [58]. 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 [59] 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 [60] 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.

Related Work

In the context of compiler optimizations for explicitly parallel languages, it is worth noting briefly the large body of research in optimizing compilers with parallel targets whose approaches are very different from ours, both to clarify our own objectives and to place this work within a broader context of work on compiler optimizations for parallel applications.

TODO: add text on related work in parallelizing compilers, HPF and other array-based parallel optimization, etc.

7.1 Optimizing Local MSA Array Accesses

7.2 Optimizing Data Exchange

Empirical studies have sometimes suggested that shared memory programming is more productive than distributed memory [61]. 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 messag-

ing, 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 [62] provided for marshalling of C structs, using a high-level

specification for communication in concert with a stub compiler. Later systems such as CORBA [63] 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 [64–66]. 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. C++2MPI [67] and the MPI Preprocessor [68] 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 [69] 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 [70]. In [71], 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 [72]. 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. Gener-

ally 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.

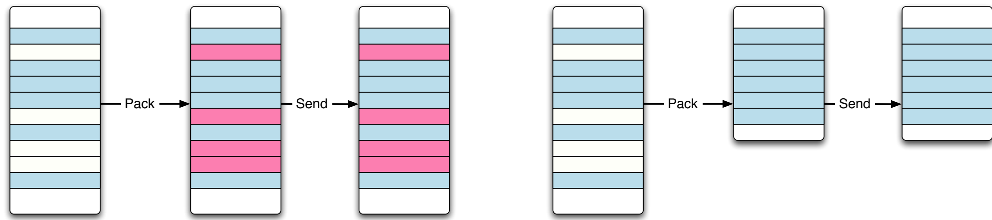
7.2.1 Implementation

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 straightforward 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.

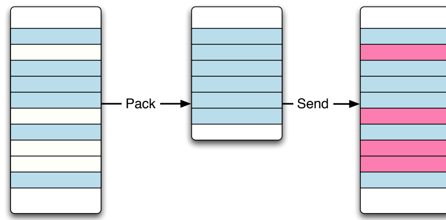
One of the benefits of our approach is that it does not require complex or time-intensive compiler analysis. For each remotely invocable method in our application, we wish to produce a function that will pack its arguments, discarding any data which can be proven to be unused. The primary question to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



(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 7.1: 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.

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 [73], 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 7.1(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.

7.2.2 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 [74], 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 7.1 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 7.2 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 is a list of particles corresponding to local particles which have forces exerted

Listing 7.1: 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 7.2: 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);
    // ...
};
```

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 are migrating to a new patch, and so none of their data can be omitted–

Listing 7.3: A pup function equivalent to the packing code Charj generates for the updateForces method.

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

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, updateParticles and updateForces 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 7.3 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 [54] from the well-known SPLASH-2 suite [75]. 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 the size of these volumes may vary greatly depending on the spatial particle

Listing 7.4: 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.

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

Listing 7.5: A Charj method signature corresponding the the method in listing 7.4.

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

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 7.4, `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 7.5 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.

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.

1. Charisma integration
2. GA abstraction
3. Examples of further optimizations

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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

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

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

[29] 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.

[30] 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.

[31] 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

[32] 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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

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

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

[44] 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.

[45] 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.

[46] 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.

[47] 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).

[48] 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

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

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

[55] L. V. Kale and S. Krishnan, "A comparison based parallel sorting algorithm," in *Proceedings of the 22nd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1993, pp. 196–200.

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

[57] *The Charm++ Programming Language Manual, (Version 6.1.3)*, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana, IL, 2010.

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

[59] 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.

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

[61] 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.

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

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

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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

[70] 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.

[71] 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.

[72] 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.

[73] 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

[74] 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.

[75] 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.