Compiler Techniques for Productive Message-Driven Parallel Programming

Aaron Becker
abecker3@illinois.edu

September 19, 2011

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.

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 [29, 24, 20, 30]. Sadly, no dramatic solution to this problem has been found, and none seems likely to present itself in the near future.

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. While this approach makes it possible to write extremely successful parallel programs, it has also created many of the productivity problems that we aim to rememdy. 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

has not been possible without a real investment of time and effort by human programmers. 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.

We believe that well-known compiler techniques can be applied to carefully targetted areas to significantly simplify the development process for high performance parallel applications without sacrificing performance. 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.

2 Approach

Proponents of new high-level programming languages have long argued that these languages can increase programmer productivity. Despite the dominance of MPI, languages such as UPC [4], Chapel [5] and X10 [7] do provide benefits in terms of elegance and concision. However, these languages often suffer due to external issues such as availability, ease of porting and integrating existing code, familiarity to application developers, and the perceived risk of using a less common approach. Despite research findings that new HPC-targeted languages can boost productivity [6, 4, 9, 14], they have thus far not been widely accepted by large-scale application developers.

However, there are existing examples of projects which leverage compiler techniques to improve programmer productivity in an HPC context by improving errors and warnings, integrating application-level features, and automating optimizations. For example, the Tensor Contraction Engine project [1] translates high-level code for describing many-body interations and uses static analysis to generate optimized code which can target different parallel programming models. While this work is domain-specific, there is also a body of work on more general applications of compiler technology to programs using the MPI, whether through improved static checking [23], improved communication optimization [19], or automated integration of application-level fault tolerance [33]. Our goal is similar in spirit: to apply compiler techniques in the context of a rich runtime system to increase productivity.

In pursuit of the goal of improved programmer productivity, we must establish a platform from which to build. Many high-level approaches to improving the experience of writing parallel code have taken a top-down approach. They first decide which features should be supported by such an effort, then design a system which can make those features available. While this approach is valid, it runs the risk of creating a platform which is irrelevant for practical use in the HPC community because the design process is divorced from the practical considerations that have guided the evolution of HPC programming.

In contrast, we have decided to take a bottom-up approach by extending a successful existing platform. This approach has multiple benefits. First, it guarantees that the underlying programming model is suitable for practical HPC applications. Second, it takes advantage of significant work already undertaken to ensure the performance and portability of the existing system. The underlying platform then provides a benchmark to compare our work against which isolates the effect of added compiler support and does not introduce confounding variables related to platform familiarity and differences in programming model.

Given this decision, we must then select a platform to extend. Two main criteria guide this selection. The first and most important criterion we used for determining a target platform is the richness and complexity of its feature set. Much of the difficulty involved in modern HPC programming comes from juggling complex resource management tasks and a wide variety of libraries and modules providing high-level features such as load balancing and fault tolerance. These features are very valuable to application developers, but their use must be paid for in complexity. We feel that managing this complex environment is one area in which programmer productivity can be greatly improved within the scope of our work, and so we aim to choose a platform that exposes a powerful but large and complex feature set to the user.

We also require our target platform to be proven in the real world. While practical viability the of our chosen platform does not affect our research agenda per se, in order for our work to have a practical impact on software development, we must choose a platform which has been used to develop useful HPC applications. Choosing a proven platform ensures that we are working with a viable and technically sound programming model. We may be able to substantially improve the experience of programming in a marginal model with the use of compiler technology, but the impact of such efforts would also be marginal.

Using these criteria, we decided to develop language and associated compiler infrastructure which targets the Charm++ runtime system. There are several reasons for selecting this platform on which to build. 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, semi-automated marshalling and unmarshalling of messages, fault tolerance, power management, use of accelerator architectures, control points, and many other modules. In addition, multiple programming models already target the Charm runtime. 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 the relatively narrow scope of MPI itself and the comparative

size and complexity of the Charm ecosystem, we claim that Charm is a better environment in which to demonstrate the merits of our approach.

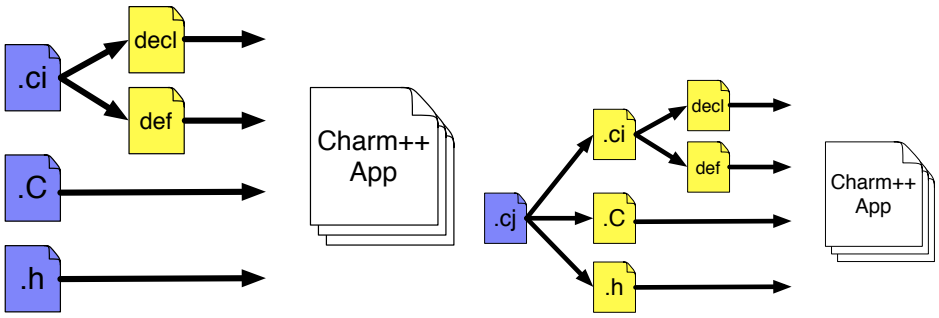
Having selected Charm as a suitable runtime system for testing our hypothesis, we are left with the question of how to demonstrate that the addition of simple compiler analysis to this system can result in improved programmer productivity without degrading performance. In particular, we must decide how to introduce compiler analysis to the system and which aspects of the programming experience we will attempt to improve. We will use our compiler infrastructure to address a wide range of questions relevant to our thesis. Specifically, what effect does model-specific syntax have on program complexity? Can we provide improved static checking by incorporating knowledge of programming model semantics into the compiler? Is there significant redundant information required in existing parallel application code that could be eliminated using compiler support? What role can a compiler play in interactions between different parallel libraries or between code written using different parallel programming models? Are there communication optimizations that can be performed automatically? Can compiler knowledge of application-level parallel features like checkpointing and load balancing lead to simpler programs?

To answer these questions in the context of Charm, we need a compiler that is aware of the Charm programming model and accompanying features. One possible approach, given that Charm is closely tied to the C++ programming language, would be to modify an existing C++ compiler to incorporate Charm-specific features and analyses. However, given the size of the C++ language and complexity of the C++ grammar, we judged this approach to be too inflexible for our purposes, especially in the case of model-specific syntax and support for interation between multiple programming models. Instead we have chosen to create a new high-level language called Charj. Our compiler infrastructure can then be used to demonstrate the utility of safety checks, optimizations, et cetera for improving productivity and performance. Creation of a new language gives us greater control over syntax and more flexibility to directly address the central questions of our thesis.

We have identified several key areas in which we will use Charj and its associated compiler infrastructure to investigate our thesis. In each of these areas, we will demonstrate advantages that come from the combination of compiler techniques with a rich parallel runtime system. The first such areas is providing increased safety and improved semantic checking for parallel-specific programming model features. The next is supporting improved syntax and support for common abstractions, thereby simplifying and shortening code without sacrificing performance or changing the underlying programming model. We also attempt to ease the process of composing independent parallel modules in an application, each potentially making use of different programming models. This ability to freely compose modules is essential to the task of simplifying the creation of large and complex parallel applications. We also aim to ease the efficient use of high-level application features like load balancing, fault tolerance, and control points. In each of these areas we will show that the addition of compiler techniques provides concrete benefits to a pre-existing programming model and runtime system.

3 Completed Work

We have already made significant strides toward demonstrating the value of simple compiler techniques in the context of a rich parallel runtime system. Here we describe the work we have already finished and the ways in which it supports our hypothesis.



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

(b) Compilation process for a Charj application.

Figure 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.

3.1 Compiler Infrastructure

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 building off of a pre-existing compiler project.

In order to minimize the time and effort spent on developing the basic infrastructure of Charj, we adopted a simple Java-like syntax and used the well-known parser generator ANTLR [25] to construct an abstract syntax tree from the input program. We perform our analysis and optimizations on this tree, then output C++ code and Charm interface code (see figure 1) which is compiled against the Charm API. 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.

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.

3.2 Model-Specific Syntax and Safety

It is often the case in discussions of programming languages that 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] http://www.haskell.org/haskellwiki/Wadlers_Law

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. While concise and powerful syntax is a major driver of programmer productivity, it can also raise significant barriers to the adoption of a language. Furthermore, many of the most tedious and most easily automated programming tasks in parallel computing have nothing to do with syntax. While syntax is undoubtedly important, in Charj we have attempted to minimize the amount of novel syntax the programmer will be exposed to by adopting a simple Java-like base syntax that adds a '@' sigil to denote proxy objects on which remote methods can be invoked and to identify remote invocations.

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

3.3 Integrating Multiple 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. 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. In particular, if one wishes to make use of parallel modules which encompass some task-specific parallel algorithm, it is extremely limiting to require that the module use the same programming model as the rest of the application.

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 interactions 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 a compiler to be aware of multiple programming models that can be used together in a single application. Such a compiler can provide much greater integration between models via improved static checking and model-specific optimizations.

One such model, already implemented on the Charm runtime, is Structured Dagger (SDAG). SDAG addresses a common need in parallel object-based applications to effectively coordinate the sequence of execution between the methods of communcating 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.

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.

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. 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.

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 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.

To address these flaws, we have integrated SDAG into the Charj infrastructure, so that SDAG syntax can be freely incorporated in any Charj function. This addresses many of the shortcomings of SDAG in a straightforward way. There is no need for SDAG methods to be separately specified and segregated in separate interface files–SDAG code is freely intermixable within a Charj application. Since the Charj compiler is SDAG-aware, there is no need for opaque atomic blocks. The compiler can directly infer these blocks and verify both that the code within them parses correctly and that it contains no forbidden parallel operations. In the case of an error, the programmer can be notified up front without relying on error messages from generated C++ code. The programmer has no need to remember to insert SDAG macros

into their datatype definitions, as this is handled transparently by the compiler. The total effect of these changes is to make SDAG much easier and less frustrating to use without altering its feature set or degrading its performance in any way.

3.4 Optimizing Data Exchange

Empirical studies have suggested that shared memory programming is more productive than distributed memory [16]. One of the factors that weighs against distributed memory programming in this analysis is the need to pack and unpack application data. Any disagreement between packing code and the corresponding unpacking code can lead to subtle bugs, and the code must be carefully maintained whenever the data being transmitted changes.

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

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

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

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

Alternatively, the programmer may use a library to assist with creating the code. This approach has the advantage that well-designed libraries can significantly ease the process of writing packing and unpacking code while increasing confidence in that code's correctness. These libraries range from the relatively spartan to full-featured libraries such as Boost.Serialization which include features for cyclic data structures and conditional packing.

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

A large amount of work has been done on data marshalling, both on improving efficiency and on reducing the burden on the programmer. Systems such as Sun RPC [31] provided for marshalling of C structs, using a high-level specification for communication in concert with a stub compiler. Later systems such as CORBA [8] 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 [10, 12, 26]. 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 [15] and the MPI Preprocessor [27]

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 [13] 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 [11]. In [32], 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 reciever 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 [18]. This library provides largely automatic support for serializing C++ data, but provides no facility for selectively omitting member data depending on context.

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

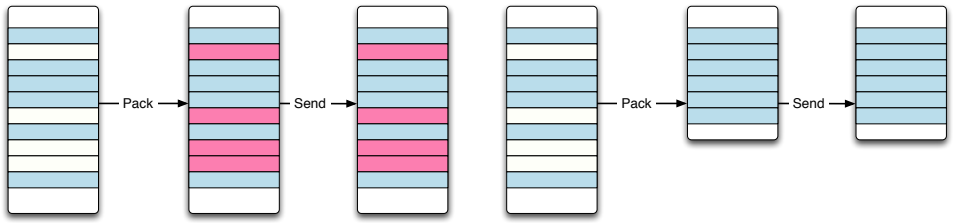
3.4.1 Implementation

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 possiblity that some fields may be needed in one situation but not in another.

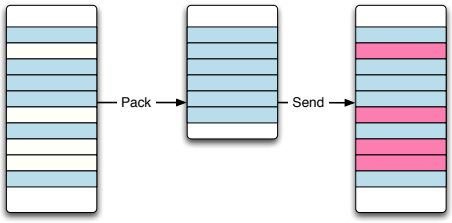
One of the benefits of our approach is that it does not require complex or time-intensive compiler analysis. For each remotely invokable method in our application, we wish to produce a function that will pack its arguments, discarding any data which can be proven to be unused. The primary question to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

Fortunately, there is a simple compiler analysis that answers this question. Since the function does not interact with its unused fields, the values in those fields are not used in any control flow



(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 2: 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.

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 [21], 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 2(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.

3.4.2 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 [3], 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 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

Listing 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);
    }
};
```

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 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 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– 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

Listing 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);
    // ...
};
```

Listing 3: A pup function equivalent to the packing code Charj generates for the `updateForces` method.

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

forces are the only particle components that can possibly be read, so method-specific packing code equivalent to listing 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 [2] from the well-known SPLASH-2 suite [28]. 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 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 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

Listing 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.

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

Listing 5: A Charj method signature corresponding the the method in listing 4.

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

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 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.

4 Planned Work

The work which we have so far completed shows the utility of close integration of compiler and runtime system in multiple areas. We will extend this work in several key areas, demonstrating the productivity benefits of compiler technology to a rich parallel runtime system in each area.

4.1 Integrating Partitioned Global Address Space Models

Above, we described the SDAG programming model and its successful integration into the Charj infrastructure. We plan to target another programming model implemented on top of the Charm runtime, Multiphase Shared Arrays (MSA), for similar integration with the goal of improving the performance of some common MSA operations transparently to the programmer. MSA is a partitioned global address space (PGAS) model, in which MSA entities all exist in a common address space while being distributed across many physical nodes. By integrating MSA into Charj, we can provide improved safety for MSA code and also perform optimizations based on the compiler's knowledge of MSA semantics.

In MSA, execution is divided into a series of phases, and each array can be accessed in one specific mode during each phase: read-only mode, in which any thread can read any element of the array; write-once mode, in which each element of the array is written to (possibly multiple

times) by at most one worker thread and no reads are allowed; accumulate mode, in which any threads can add values to any array element, and no reads or writes are permitted; and owner-computes mode, where a specified function is performed on each distributed block of the array by a processor on its home node. A synchronization call is used to move from one phase to the next.

MSA is implemented as a C++ library which is tightly integrated with the Charm runtime. It makes clever use of the C++ type system and template functionality to enforce some aspects of the programming model, relying on runtime checking for the remaining cases. The lack of special syntax in MSA makes it easy to mistake remote array operations for local array accesses, because they are syntactically identical. MSA's mechanism for enforcing semantics through the C++ type system is inflexible and difficult to extend beyond the simple cases of read-only and write-only access. In addition, because there is no distinction in the MSA programming model between elements stored locally and elements which must be fetched from another node, simple array accesses are inefficient because they must first verify that the element to be accessed is locally available. In cases where there is a regular pattern of array accesses it is much more efficient to fetch all the elements to be accessed, then perform all the accesses without performing any checking. However, due to MSA's library implementation this approach is quite cumbersome and forgoes even the limited syntactic sugar that C++ operator overloading provides to protected array accesses. This lengthens and complicates the code unnecessarily and discourages high performance code in favor of a simpler but less efficient expression.

By integrating MSA with Charj, we can take advantage of the opportunity to provide improved semantic checks at compile time while providing improved syntax. There is also scope for beneficial optimizations that will allow programmers to write simple code while attaining the performance associated with much more complex code.

One example is the conversion of safe, checked array accesses which may access either local or remote elements in a loop to a preamble which assures the locality of all accessed elements followed by purely local unchecked array accesses in the loop body. Currently in MSA this type of optimization requires abandoning convenient syntax and substantially lengthening and complicating simple loops, but in Charj we aim to provide this transformation automatically so that the simple and direct expression of a loop body by the programmer is translated into an efficient if somewhat more complicated implementation.

4.2 Accelerator-Based Models

The relative power efficiency and high peak FLOPs of floating point accelerator architectures, particularly GPUs, has made them an increasingly common target for HPC applications. However, this power comes at the cost of a more restrictive programming model, increasing the difficulty of developing high performance code for these platforms. Significant work has already been done to facilitate the use of these models by programs which use the Charm runtime [17, 22]. However, the use of these models requires the programmer to obey strict rules about read and write access to device buffers. These rules are not enforced by the compiler, and violating them can lead to subtle runtime errors that require significant time and effort to debug. In addition, the use of accelerators in Charm applications requires all accelerator code to be segregated in interface files, and exposes some implementation details of the Charm runtime, thereby raising the barrier to entry for new Charm programmers.

These problems can be ameliorated by the use of the Charj compiler. We can detect illegal reads and writes to device buffers and warn the programmer at compile-time. We can also actively identify functions which could be accelerated without requiring the programmer to explicitly designate that a function can be accelerated. Although it is straightforward to determine whether or not it is possible to run a given function on an accelerator (simply check

all code reachable from the function and ensure that all operations that could be performed are legal on the desired accelerator device), it is more complicated to determine whether or not it is *desirable* to do so. Some of the factors controlling this decision are local, such as the FLOP to memory access ratio, number of potential branches in the code, and function grain size. There are also global considerations, such as the total balance of CPU work to accelerator work in the application and whether or not the function is on the application's critical path.

In general, there is not enough information available at compile time to determine whether or not it will be advantageous to run a particular function on accelerator hardware. But while we cannot effectively make the decision, we can both make it easier for the programmer to indicate what should be done and facilitate the collection of data at runtime which can improve our ability to choose between CPU- and accelerator-based execution. To facilitate programmer choice between architectures we simply require a single keyword that indicates that a function is intended to run on an accelerator. This allows the compiler to verify that the code to be accelerated can run on the intended target and generate the appropriate code to manage the accelerator-based exectution. We can also gather data at runtime on function grain size and identify an application's critical paths and use this information to guide our choice of which execution resources to use.

In addition, we can restructure user code transparently to avoid current limitations set on accelerated code. For example, functions which use the Charm accelerator framework cannot invoke entry methods. They can only return data via a callback which is implicitly invoked at the termination of the funcion. Using Charj, we can provide the ability for the programmer to invoke entry methods in the middle of an accelerated method in cases where there are no dependencies between the output data and the remainder of the accelerated method simply by agglommerating these invocations and automatically constructing a callback to issue them at the end of the accelerated method.

4.3 User-Level Threads

In the base Charm programming model, objects can be migrated safely as long as they are not in the midst of executing an entry method. However, other programming models that use the Charm runtime and interoperate with Charm programs do not necessarily have this property. For example, Adaptive MPI (AMPI) programs implement the message passing model of MPI on top of the Charm runtime, but AMPI threads cannot be safely migrated without a scheme for migrating the stack of blocked AMPI processes. Several such schemes exist. Some explicitly heap-allocate stacks and manage them at the user level, others make use of the executable's global offset table to manage the migration, and still others carefully control the allocation of memory across all nodes of a cluster in order to facilitate transfers between nodes.

We believe that we can use compiler analysis to improve the state of thread migration on the Charm runtime. The compiler can provide guesses at needed stack sizes, keep track of reachable heap memory and rewrite heap pointers as a part of the thread migration process. This can mitigate the need for programmers to make a guess at what stack size their application will require, with incorrect results if they guess too low and wasted memory and lowered performance if they guess too high. By providing support in this area, Charj can increase the scope for programming models to interoperate seamlessly by facilitating the migration of blocked threads with an associated stack.

4.4 Addressing Shared Address Space Programming

Increasingly, HPC programs execute in a multicore environment. Though the number of nodes in large clusters is always growing, the number of cores per node is growing at the same time.

Although applications can ignore the fact that within a node processes can share memory, by doing so they miss a significant opportunity to reduce communication and memory overhead within nodes. The growing importance of shared memory programming within a distributed memory cluster has led to the rise of such hybrid programming models as MPI+OpenMP, in which MPI handles inter-node communication while parallelism within a node is managed by OpenMP. Similarly, Charm programs must adapt to remain competitive, and doing so will require tools for exploiting shared memory safely without sacrificing performance.

For example, a small library called Boomerang has been developed to facilitate the use of shared memory within Charm applications. This library will transparently send data to objects on remote nodes or send just a pointer to a shared buffer to objects that reside on the same node as the sender. However, to ensure correctness the programmer must carefully manage the accesses to the potentially shared data to avoid conflicts. The burden of deciding who can access which data when is left solely to the programmer's determination, and there is no mechanism for enforcement or notification when data race errors occur.

We will formulate a more sophisticated version of this functionality in Charj which allows the programmer to express the types of access that each program entity has at each phase of the program while enforcing those rights and notifying the programmer of any conflicts. These rights would apply not just to individual variables or entire collections, but also to subsets of collections. For example, in a parallel quicksort implementation a process might delegate its read and write capabilities to a recursively spawned subprocess that is responsible for sorting half of the current array. The delegating process cannot modify memory whose rights it has delegated until it reclaims those rights. These capabilities will facilitate effective, safe use of shared memory within the Charm programming model.

4.5 Reducing Memory Footprint with Live Variable Analysis

Some operations common in parallel applications require knowledge of the current set of live data in an application. One example is checkpointing, in which application state is preserved so that in the event of a later crash the application can be rolled back to a known consistent state. Another example is process migration, in which one process of a parallel application is moved from one physical node to another, often in service of load balancing. In these cases it is sufficient to merely store all of the memory allocated by a process, whether or not it remains live. However, this can be wasteful. One common alternative is for the programmer to explicitly list the state that should be preserved, but this is work-intensive and error-prone, especially as applications evolve, introducing new variables and using old ones in new ways.

Using live variable analysis, the compiler can identify the set of live memory and checkpoint or migrate only that data which can actually be accessed later. This produces reliable, efficient code without programmer intervention. Such an approach has been used successfully as a standalone tool for MPI applications [33]. Here we can integrate the functionality more closely into the programming environment and provide this functionality in multiple contexts.

4.6 Support for High-level Features

A large amount work on the Charm runtime is dedicated not to core runtime features or application development, but rather to the development of high-level features which can be adopted by applications to provide services that provide utility to a variety of HPC applications. For example, the load balancing framework provides a load measurement facility and a suite of algorithms for migrating Charm objects, making it easy for Charm applications to implement measurement-based load balancing.

Similarly, Charm has functionality to assist with fault tolerance, checkpointing, runtime optimization via control points, and other application-level features. In some cases it may be possible to assist the programmer in making use of these features via improved syntax or automated analysis. We will investigate the potential uses of Charj to make it easier for programmers to use these high-level features.

5 Summary

HPC application developers face many difficulties that impede the productive construction of high performance, scalable applications. Although some of these difficulties are inherent to the field, others could be prevented by automated tools. Feature-rich parallel runtime systems which can provide many of these tools already exist, but learning the conventions and semantics of these runtime systems is a difficult task in itself, providing a significant barrier to their adoption.

It is our view that a rich runtime system can be combined with simple compiler technology to ease the programming process and increase productivity without compromising performance. A compiler can enforce programming model semantics and provide useful warnings and error messages in a way that library-based tools cannot, as with the Charj compiler's enforcement of Charm "readonly" data rules. It can ease the development of multi-model programs, as with SDAG. And it can automatically perform optimizations that would be tedious and error-prone to perform by hand, as with the automatic generation of optimized packing and unpacking functions. In the coming months we will demonstrate further areas in which simple compiler support can simplify the lives of HPC programmers and attempt to quantify the advantages of this simplification.

References

[1] Er A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Hya Krishnan, Chi chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, and Er Sibiryakov. Automatic code generation for many-body electronic structure methods: The tensor contraction engine. *Molecular Physics*, 104, 2006.

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

[3] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. NAMD: A Portable and Highly Scalable Program for Biomolecular Simulations. Technical Report UIUCDCS-R-2009-3034, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2009.

[4] Francois Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity analysis of the upc language. In *18th International Parallel and Distributed Processing Symposium*, page 254, 2004.

[5] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[6] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

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

[9] DARPA. High Productivity Computing System (HPCS) Industry Study: Proposer Information Pamphlet. http://www.darpa.mil/ipto/solicitations/closed/02-09_PIP.htm.

[10] 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*, pages 40 –47, 1998.

[11] D. Dig 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.

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

[13] Delphine Goujon, Martial Michel, Jasper Peeters, and Judith Devaney. Automap and autolink tools for communicating complex and dynamic data-structures using mpi. In Dhabaleswar Panda and Craig Stunkel, editors, *Network-Based Parallel Computing Communication, Architecture, and Applications*, volume 1362 of *Lecture Notes in Computer Science*, pages 98–109. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0052210.

[14] H. Wen and S. Sbaraglia and S. Seelam and I. Chung and G. Cong and D. Klepacki. A Productivity Centered Tools Framework for Application Performance Tuning. In *QEST '07: Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems*, pages 273–274, Washington, DC, USA, 2007. IEEE Computer Society.

[15] 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. PAR-ELEC 2000. Proceedings. International Conference on*, pages 13 –17, 2000.

[16] Lorin Hochstein and Victor 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*, SPAA '06, pages 114–114, New York, NY, USA, 2006. ACM.

[17] Laxmikant V. Kale, David M. Kunzman, and Lukasz Wesolowski. Accelerator Support in the Charm++ Parallel Programming Model. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore and Accelerators*, pages 393–412. CRC Press, Taylor & Francis Group, 2011.

[18] Prabhanjan Kambadur, Douglas Gregor, Andrew Lumsdaine, and Amey Dharurkar. Modernizing the c++ interface to mpi. In Bernd Mohr, Jesper Traff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 266–274. Springer Berlin / Heidelberg, 2006.

[19] Amit Karwande, Xin Yuan, and David K. Lowenthal. Cc–mpi: a compiled communication capable mpi prototype for ethernet switched clusters. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 95–106, New York, NY, USA, 2003. ACM.

[20] Jeremy Kepner. Hpc productivity: An overarching view. *International Journal of High Performance Computing Applications*, 18(4):393–397, Winter 2004.

[21] Lawrence T. Kou. On live-dead analysis for global data flow problems. *J. ACM*, 24:473–483, July 1977.

[22] David M. Kunzman and Laxmikant V. Kalé. Towards a framework for abstracting accelerators in parallel applications: experience with cell. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[23] Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. Mpi-check: a tool for checking fortran 90 mpi programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[24] Thomas Panas, Dan Quinlan, and Richard 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*, SE-HPC '07, pages 2–, Washington, DC, USA, 2007. IEEE Computer Society.

[25] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[26] Christian Queinnec. Marshaling/demarshaling as a compilation/interpretation process. *Parallel Processing Symposium, International*, 0:616, 1999.

[27] 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*, pages 7 pp. –256, 0-0 2006.

[28] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[29] Marc Snir and David A. Bader. A framework for measuring supercomputer productivity. *International Journal of High Performance Computing Applications*, 18(4):417–432, Winter 2004.

[30] Thomas Sterling. Productivity metrics and models for high performance computing. *International Journal of High Performance Computing Applications*, 18(4):433–440, Winter 2004.

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

[32] 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*, pages 1 –12, april 2008.

[33] Xuejun Yang, Panfeng Wang, Hongyi Fu, Yunfei Du, Zhiyuan Wang, and Jia Jia. Compiler-assisted application-level checkpointing for mpi programs. *Distributed Computing Systems, International Conference on*, 0:251–259, 2008.