© Copyright by Chao Huang, 2007

SUPPORTING MULTI-PARADIGM PARALLEL PROGRAMMING ON AN ADAPTIVE RUN-TIME SYSTEM

BY

CHAO HUANG

B.Eng., Tsinghua University, 2001
M.S., University of Illinois at Urbana-Champaign, 2004

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, 2007

Urbana, Illinois

Abstract

Recent developments in supercomputing have brought us massively parallel machines. With the number of processors multiplying, the appetite for more powerful applications that can take advantage of these large scale platforms has never ceased growing. Modern parallel applications typically have complex structure and dynamic behavior. These applications are composed of multiple components and have interleaving concurrent control flows. The workload pattern of these applications shifts during execution, causing load imbalance at run-time. Programming productivity, or the effectiveness and efficiency of programming high performance applications for these parallel platforms, has become a challenging issue.

One of the most important observations during our pursuit of high productivity with scalable performance for complex and dynamic parallel applications was that adaptive resource management can and should be automated. The PPL research group has developed an Adaptive Run-Time System (ARTS) and a parallel programming language called Charm++ for automatic resource management via migratable objects.

There are two obstacles in our pursuit of high productivity with the ARTS. The first is effective expression of global view of control in complex parallel programs. Traditional paradigms such as MPI and Global Address Space (GAS) paradigms, although popular, suffer from a drawback in modularity. For applications with multiple modules, they do not allow the runtime control over resource management of individual modules. Although Charm++ provides resources management capabilities and logical separation of multiple modules, its object-based message-driven model tends to obscure the global flow of control. We will explore new approaches to describing the flow of control for complicated parallel applications. As a reference implementation, we introduce a

language *Charisma* for expressing the global view of control that can take advantage of the ARTS. We carry out productivity and performance study of Charisma, with various examples and real-life applications.

The second issue is to efficiently accommodate existing prevalent programming paradigms. Different programming models suit different types of algorithms and applications. Also the programmer proficiency and preference may result in the variety of choices of programming languages and models. In particular, there are already many parallel libraries and applications written with prevalent paradigms such as MPI. In this thesis, we explore research issues in providing adaptivity support for prevalent paradigms. We will evaluate important existing parallel programming languages and develop virtualization techniques that bring the benefits of the ARTS to applications written using them. As a concrete example, we evaluate our implementation of *Adaptive MPI* in the context of benchmarks and applications.

As applications grow in size, their development will be carried out by different teams with different paradigms, to best accommodate the expertise of the programmers and the requirements of the different application components. These paradigms include the new paradigm as represented by Charisma's global description of control, as well as existing ones such as MPI, GAS and Charm++. Charm++'s adaptive run-time system is a good candidate for a common environment for these paradigms to interoperate, and this thesis demonstrates the effectiveness of our research work for interoperability across multiple paradigms. The ultimate goal is to unify these various aspects and support multiparadigm parallel programming on a common run-time system for next-generation parallel applications.

Acknowledgments

First and foremost, I thank my advisor Professor Laxmikant (Sanjay) Kalé, for his persistent inspiration and continuous guidance. During my six years at PPL, I have learned not only how to do research and excel at school, but also how to cooperate and coordinate in such a big team as our group. It definitely adds to my invaluable personal asset and will benefit my career and life in the future.

I thank my dissertation committee, Professor David Padua, Professor Marc Snir, Professor Ponnuswamy (Saday) Sadayappan and Professor Maria Garzaran, for their helpful advice and suggestions.

I thank my colleagues at PPL, whose names seem too long to list here but I will give it a try. I thank Gengbin Zheng, Eric Bohm, Terry Wilmarth, Celso Mendes, Chee Wai Lee, Sayantan Chakravorty, Filippo Gioachin, Pritish Jetley, David Kunzman, Isaac Dooley, Abhinav Bhatele, and Aaron Becker, as well as former PPLers, Sameer Kumar, Milind Bhandarkar, Orion Lawlor, Greg Koenig, Jayant DeSouza, Yan Shi, and Mark Hills. It is my great honor to work (and play) with such a brilliant group of people.

I thank all my friends at Urbana-Champaign. You made life in the corn field much more interesting than it may sound. All the parties, games, dinners and excursions saved me from becoming a nerd with Permanent Head Damage.

Most of all, I thank my family. My parents are most proud of me even though they have absolutely no idea what or how I have been doing in my thesis research. I miss my late grandmother Rui Qian, whose love and spirit drives me all along the way. I thank my newborn son David Rui Huang for giving me a higher appreciation for life. Your arrival might have slowed down my thesis

progress a little bit, but every time I think of you, I feel 110% motivated. Last but not least, I thank my wonderful wife, Jia Guo, for always being there for me. Her unconditional love is simply beyond the description of my language (no pun intended).

Table of Contents

List of Tables . . . . **xii**

List of Figures . . . . **xiv**

Chapter 1 Introduction **1**
1.1 Thesis Contributions 3
1.2 Thesis Organization 4

Chapter 2 Multi-Paradigm Parallel Programming Support **7**
2.1 Charm++ and Its ARTS 7
2.2 Supporting Multi-Paradigm Programming on a Common ARTS 10
2.3 Approach and Objectives 11
2.3.1 Thesis Objectives 12
2.3.2 Proposed Architecture 13

Chapter 3 Toward a Productive Parallel Programming Language **15**
3.1 A Motivating Example 15
3.2 Design Goals 20
3.2.1 Higher Level of Abstraction 20
3.2.2 Separation of Parallelism Specification and Sequential Component Development 21
3.2.3 Interoperability with ARTS 22
3.2.4 Proposed Parallel Programming Paradigm 22

Chapter 4 Charisma: Orchestrating Migratable Parallel Objects **25**
4.1 Language Design 25
4.1.1 Parallel Object Array 26
4.1.2 `Foreach` Statement 28
4.1.3 Producer-Consumer Model 29
4.1.4 Data Dependence and Program Order 30
4.1.5 Program Determinacy 32
4.1.6 Describing Communication Patterns 33
4.1.7 Sequential Code 37
4.2 Library Module Development with Charisma 38
4.2.1 Parallel Library Interfaces 39
4.2.2 Charisma Module Support 40

4.2.3 Library Support for Charm++ . . . 44
4.3 Implementation Issues . . . 45
4.3.1 Dependence Analysis . . . 45
4.3.2 Control Transfer . . . 46
4.3.3 User Code Integration . . . 47
4.3.4 Generated Code Optimizations . . . 47
4.4 Extensions, Restrictions and Limitations . . . 48
4.4.1 Overlap Extension . . . 48
4.4.2 Limitations of Charisma . . . 50
4.5 Related Work . . . 52

Chapter 5 Evaluation of Charisma . . . 55
5.1 Performance Evaluation . . . 55
5.1.1 Stencil Calculation . . . 55
5.1.2 3D FFT . . . 57
5.1.3 Wator . . . 58
5.2 Classroom Productivity Study . . . 60
5.2.1 Experiment Environment and Results . . . 61
5.2.2 Productivity Analysis . . . 63
5.3 Code Comparison: MD . . . 65

Chapter 6 Charisma Application Case Study . . . 69
6.1 LeanCP . . . 69
6.1.1 Implementation with Charisma . . . 71
6.1.2 Results . . . 75
6.2 Parallel Topology Optimization . . . 76
6.2.1 Development Process . . . 77
6.2.2 Results . . . 79

Chapter 7 Adaptivity Support for Prevalent Languages . . . 83
7.1 Design Goals . . . 84
7.2 Processor Virtualization Via Migratable Threads . . . 85
7.2.1 Charm++ Facilities . . . 85
7.2.2 Implementing Virtual Processes . . . 86
7.2.3 Handling Global Variables . . . 88
7.2.4 Migrating Thread Data . . . 89
7.2.5 Automatic Checkpointing . . . 90
7.3 Adaptive MPI . . . 91
7.3.1 Support for Sequential Replay of an MPI Node . . . 93
7.4 Adaptive Implementation of ARMCI . . . 94
7.4.1 Performance Evaluation . . . 95
7.5 Interoperability Support . . . 97
7.5.1 Inter-Module Interoperability . . . 97
7.5.2 Inter-Paradigm Interoperability . . . 98
7.6 Related Work . . . 99

Chapter 8 Evaluation of Adaptive MPI . . . 101
8.1 AMPI Performance Evaluation . . . 101
8.1.1 Virtualization Overheads . . . 102
8.1.2 Flexibility to Run . . . 103
8.1.3 Adaptive Overlapping . . . 104
8.1.4 Automatic Load Balancing . . . 107
8.1.5 Checkpoint Overhead . . . 108
8.2 Application Case Study . . . 110
8.2.1 Rocstar . . . 110
8.2.2 Fractography3D . . . 113

Chapter 9 Conclusions . . . 117

Appendix A Charisma Manual . . . 121
A.1 Charisma Syntax . . . 121
A.1.1 Orchestration Code . . . 121
A.1.2 Sequential Code . . . 128
A.2 Building and Running a Charisma Program . . . 130
A.3 Support for Library Module . . . 131
A.4 Writing Module Library . . . 131
A.5 Using Module Library . . . 131
A.6 Using Load Balancing Module . . . 132
A.6.1 Coding . . . 132
A.6.2 Compiling and Running . . . 133
A.7 Handling Sparse Object Arrays . . . 133

Appendix B LeanCP Orchestration Code . . . 135
B.1 Header Section . . . 135
B.2 Declaration Section . . . 136
B.3 Orchestration Section . . . 136

Appendix C AMPI Extension API . . . 139
C.1 Running with Virtual Processes . . . 139
C.2 Automatic Load Balancing Interface . . . 139
C.3 Automatic Checkpointing Interface . . . 140
C.4 Asynchronous Collective Communication Interface . . . 140

References . . . 143

Author's Biography . . . 155

List of Tables

5.1 SLOC Comparison of Wator 60
5.2 Number of Hours Spent on Development (Sample size 19) 61
5.3 Percentage of Development Time Reduction Using Charisma over Charm++ (Sample size 19) 62
5.4 Percentage of Development Time Reduction Using Charisma over Charm++ (Graduate vs. Undergradu
5.5 Percentage of Development Time Reduction Using Charisma over Charm++ (CS vs. Non-CS) 62

8.1 Timestep Time [ms] of 240^3 3D 7-point Stencil Calculation with AMPI vs. Native MPI on Lemieux103
8.2 Timestep Time [ms] of 960^3 3D 7-point Stencil Calculation with AMPI v.s. Native MPI on NCSA IA-6
8.3 Iteration Time [ms] of K^3 3D 7-point Stencil Calculation on 8 PEs of NCSA IA-64 Cluster105
8.4 *Rocstar* Performance Comparison of 480-processor Dataset for Titan IV SRMU Rocket Motor on Appl

List of Figures

2.1 Parallel Programming Based On Migratable Objects 8
2.2 Message-Driven Execution With a Processor-Level Scheduler 9
2.3 Architecture Supporting Multi-Paradigm Parallel Programming on the ARTS . . . 13

3.1 Structure of a Molecular Dynamics Simulation Application: NAMD 17

4.1 Flowchart of Charisma 27
4.2 Charisma Orchestration Code Example: Parallel Object Arrays 27
4.3 Charisma Orchestration Code Example: `foreach` Statement 28
4.4 Charisma Orchestration Code Example: Publish Statement 29
4.5 Charisma Orchestration Code Example: Parameter Variables 30
4.6 Charisma Orchestration Code Example: Program Order 31
4.7 Charisma Orchestration Code Example: MD Example 32
4.8 Charisma Orchestration Code Example: Program Order with Loops 33
4.9 Charisma Orchestration Code Example: Point-to-Point Communication 34
4.10 Charisma Orchestration Code Example: Reduction 35
4.11 Charisma Orchestration Code Example: Multicast 35
4.12 Charisma Orchestration Code Example: Scatter 36
4.13 Charisma Orchestration Code Example: Gather 36
4.14 Charisma Orchestration Code Example: Permutation Operation 37
4.15 Charisma Sequential Code Example: Outport and Producing Values 38
4.16 Charisma Sequential Code Example: Reduction 38
4.17 Interaction Patterns for Parallel Library Modules 40
4.18 Library for Charisma: 3D FFT Example 41
4.19 Library for Charisma: Using 3D FFT Library 42
4.20 Library for Charisma: Using Multiple Instances of 3D FFT Library 43
4.21 Library for Charm++: `start` 44
4.22 Library for Charm++: `done` 45
4.23 Charisma Orchestration Code Example: Overlap Statement on Different Objects . . 49
4.24 Charisma Orchestration Code Example: Overlap Statement on Same Object 50

5.1 Performance of Stencil Calculation 56
5.2 Charisma Overhead Breakdownm 57
5.3 Transpose-based 3D FFT Algorithm 58
5.4 Performance of 3D FFT 58
5.5 Orchestration Code for 3D FFT 58

5.6 Screenshot of Realtime Visualization of Wator 59
5.7 Spread Plot of Development Time with Charisma vs. Charm++ 63
5.8 MD with Charisma: Clear Expression of Global View of Control 66
5.9 MD with Charm++: Overall Control Flow Buried in Objects' Code 66
5.10 MD with MPI: Additional Code Required for Performance 67

6.1 Visualization of Human Carbonic Anhydrase. The cloud in the wire frame represents the electron densi
6.2 Structure of LeanCP 72
6.3 Point-to-Point Operation in LeanCP 74
6.4 Transpose and Reduction Operations in LeanCP 75
6.5 Multicast and Transpose Operations in LeanCP 75
6.6 Performance of Charisma Version of LeanCP on Turing Cluster 76
6.7 Topology Optimization Process 78
6.8 Visualization of Optimized Topology for 3D Heat Transfer Problem 80
6.9 Performance of 1,000,000 Element Topology Optimization Application on Turing Cluster 81

7.1 Implementation of Virtual Processors 87
7.2 Migrating a Thread Stack Allocated with Isomalloc 90
7.3 Structure of Rocket Simulation Code with Typical MPI Implementation 92
7.4 Structure of Rocket Simulation Code with AMPI's Adaptivity Support 92
7.5 Contiguous Copy Performance of Adaptive and Native Implementations 96
7.6 Strided Copy Performance of Adaptive and Native Implementations 96
7.7 Example of Creating Charm++ Objects in an AMPI Program 99

8.1 Point-to-point Performance on NCSA IA-64 Cluster 102
8.2 Point-to-point Communication Time 103
8.3 Performance of Ping-pong vs Multi-ping Benchmark on Turing (Apple G5) Cluster 105
8.4 7-point Stencil Timeline with 1, 2 and 4 VPs Per Processor 106
8.5 Load Balancing on NAS BT-MZ 108
8.6 Checkpoint Overhead of NAS Benchmark on Turing Cluster 109
8.7 Titan IV Propellant Slumping Visualization 111
8.8 Fractography3D: Crack Propagation Visualization 113
8.9 CPU Utilization Projections Graph of Fractography3D Over Time With and Without Load Balancing11
8.10 CPU Utilization Graphs of Fractography3D Across Processor With and Without Load Balancing114

Chapter 1

Introduction

Recent developments in supercomputing have brought us massively parallel machines. Even with the number of processors multiplying, the appetite for more powerful applications that can take advantage of these large scale platforms has never ceased growing. Programming productivity, or the effectiveness and efficiency of programming high performance applications for these parallel platforms, has become a challenging issue.

Modern parallel applications typically have complex structure and dynamic behavior. These applications are composed of multiple components and may have concurrent control flows. The workload pattern of these applications shifts during execution, causing load imbalance at run-time.

A good example is the rocket simulation code developed at the Center for Simulation of Advanced Rockets (CSAR). The focus of the code is the accurate physical simulation of solid-propellant rockets, such as the Space Shuttle's solid rocket boosters [1, 2]. One version of the main CSAR simulation code consists of four major components: a fluid dynamics simulation, for the hot gas flowing through and out of the rocket; a surface burning model for the solid propellant; a non-matching but fully-coupled fluid/solid interface; and finally a finite-element solid mechanics simulation for the solid propellant and rocket casing. The simulation exhibits a dynamic nature and the MPI model is not always able to handle it well. For instance, as the solid propellant burns away, each processor's portion of the problem domain changes, which will change the CPU and communication time required by that processor. Moreover, the simulator's main loop consists of one call to each of the simulation components in turn, in a one-at-a-time lockstep fashion. This

means, for example, the fluid simulation must finish its timestep before the solids can begin its own with the current implementation with MPI model. Clearly, the application developers need paradigms and tools that better accommodate the dynamic nature of their algorithms.

On the other hand, next-generation supercomputing platforms keep growing in size and complexity. For example, Blue Gene/L [3] by IBM has 64K dual processor nodes, scoring over 280 teraflops sustained performance. With the large number of processors and physical cabling limitations, the main communication interconnect is organized into a 3D torus. This means a relatively smaller cross-section bandwidth and more hops across the machine, which requires extra resource management attention in the programs. The forces driving these advances will continue to grow. DARPA High Productivity Computing Systems Program [4, 5] has set high goals for building next generation supercomputing systems. Its program mission puts equal stress on performance and productivity.

- **Performance:** Improve the computational efficiency and performance of critical national security applications by 10X to 40X over today's scalable vector and commodity high performance solutions for systems comprised of ten's to thousands of computing nodes.

- **Productivity:** Reduce the cost of developing, operating, and maintaining HPCS application solutions.

Several major vendors have been designing more aggressive architectures for supercomputers to be built within the next 5-10 years. Many of them take advantage of system-on-chip and multi-core technology. In terms of programmability, multi-core exposes much of the underlying hardware to the programmer, allowing the development of very high performance, finely tuned software. However, this comes at a price of increased difficulty of efficient programming, posing greater productivity challenges to parallel programming.

1.1 Thesis Contributions

This thesis explores new research directions in supporting multi-paradigm parallel programming on an Adaptive Run-Time System (ARTS). An ARTS can enhance productivity by automating dynamic resource management in a parallel program [6]. We derive our research objectives from challenges and practical issues encountered in our research on achieving high productivity and performance for a wide range of parallel applications. Firstly, we investigate a new paradigm of programming global control flow via object-level orchestration. For this purpose we develop a reference language called Charisma, and study the productivity benefit of various techniques. Secondly, because prevalent paradigms capture a significant number of existing programs, we investigate the techniques of supporting these paradigms on our adaptive run-time system. To verify the advantages of adaptivity support for these paradigms with a common run-time system, we created adaptive implementations of MPI and ARMCI standards. The implementations exhibit performance benefits over benchmarks and applications. The interoperability across the underlying run-time offers productivity benefits for developing large-scale multi-paradigm applications. In short, our research work has achieved the desired effects and aims at inspiring further research in the area of parallel productivity.

The thesis makes several contributions to the support of multi-paradigm programming with both improved productivity and optimized performance, primarily in the following aspects.

- **New paradigm that allows more efficient collaboration in parallel programming:** From our collaborations with scientists and engineers to develop parallel programs, we observed some productivity issues and accordingly proposed a new paradigm and programming pattern with separation of parallelism specification, to achieve more efficient collaboration in parallel programming.

- **A high level parallel language, Charisma:** Charisma offers higher-level abstraction in describing global view of control in parallel program. It also separates sequential components development from parallel flow organization. The language is targeted for novice paral-

lel programmers with scientific and engineering background, and its productivity advantage over Charm++ is demonstrated using a small classroom study. Furthermore, Charisma facilitates library development and reuse for both Charisma and Charm++ programs on the ARTS.

- **Adaptive implementations of MPI and ARMCI:** Adaptivity support for existing prevalent paradigms not only makes the performance benefits of Charm++'s run-time available to a wider range of existing applications, but also enables future large scale parallel applications to be built across multiple paradigms on top of the same run-time system.

- **Interoperability:** Programs built with both Charisma and our implementations of MPI and ARMCI are capable of interoperating on the common adaptive run-time system, as other Charm++ programs do. Our system facilitates cross-paradigm development, such as reusing a Charisma library in a Charm++ program, or using a Charm++ module in an AMPI-based application.

1.2 Thesis Organization

Chapter 2 overviews supporting multiparadigm parallel programming on a common run-time system. We describe Charm++ and its ARTS with their novel features and performance benefits. Then we explain the motivation behind this thesis's topic: supporting multi-paradigm parallel programming on the ARTS.

Chapter 3 introduces our research exploration in an effort to design a high productivity language for parallel programming. We show a motivating example for our search for a high-level language that allows global view of control expression, and explain our design goals for the new language, through analysis of the typical process of parallel program development.

In Chapter 4, we introduce our new high level language called *Charisma*. Charisma is designed to allow the programmer to describe the global flow of control and simultaneously specify logical

separation between modules for purposes of resource management in an object-based parallel program. We explain various design and implementation issues of Charisma, as well as the challenges of defining the library interface with Charisma.

Chapter 5 contains an evaluation of Charisma, in terms of both performance and productivity. Along with comparing scaling performance of benchmarks with Charisma vs. Charm++, we present the results and analysis of a classroom study of Charisma's productivity. This Chapter ends with a concrete example of molecular dynamics coded in different languages, to further illustrate the productivity advantage of Charisma.

In Chapter 6, we showcase two examples of Charisma being used in complex applications: the quantum chemistry simulation *LeanCP*, and topology optimization *topt*.

Chapter 7, we examine the existing parallel programming paradigms and explain research challenges we face in supporting adaptivity for them. As examples, we present our implementation of two important models: MPI for message passing and ARMCI for global address space. Detailed performance analysis and application showcases of Adaptive MPI are presented in Chapter 8.

Finally, Chapter 9 concludes the thesis and proposes future work.

Chapter 2

Multi-Paradigm Parallel Programming Support

This thesis is focused on means for achieving high productivity in parallel programming. Our approach is to support productive multi-paradigm programming on top of an Adaptive Run-Time System (ARTS). Before launching into a discussion of the benefits of having a common run-time system and our proposed architecture, we first give a brief introduction to the Charm++ parallel programming language and its ARTS.

2.1 Charm++ and Its ARTS

At the Parallel Programming Laboratory (PPL), our approach to parallel programming strives to achieve an optimal division of labor between the run-time system and the programmer. In particular, it is based on the idea of migratable objects. The programmer decomposes the application into a large number of parallel computations executed on parallel objects, while the run-time system assigns those objects to processors (Figure 2.1). This approach gives the run-time system the flexibility to migrate objects among processors to effect load balance and communication optimizations.

Charm++ [6, 7] is an object-based parallel programming language that embodies this concept. A Charm++ program consists of parallel entities, either objects called *chares* or light-weight user-level threads. Many of these objects can be organized into an indexed collection, called a *chare*

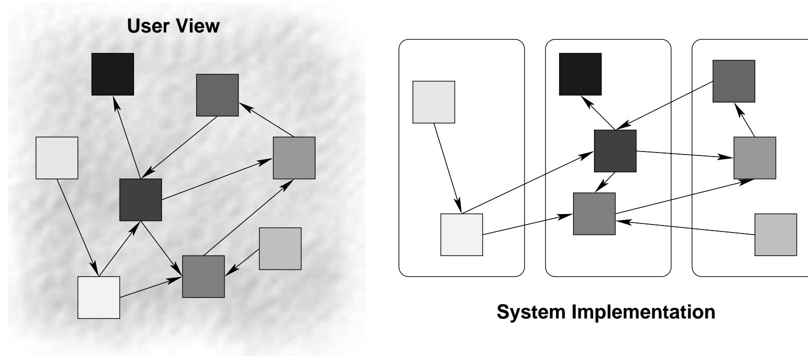


Figure 2.1: Parallel Programming Based On Migratable Objects

array. A chare array can be 1-D or multi-dimensional, dense or sparse. The index on the object array can be as flexible as user-defined bit patterns. For instance, an object array indexed via a bit-vector can be organized to represent the structure of a tree. The underlying run-time system typically maps multiple parallel objects onto one physical processor and moves the migratable objects across processors as needed.

Execution on an object is triggered when another object sends a message targeting the recipient object's entry point or a specially registered function for remote invocation. Note that the remote invocation is asynchronous: it returns immediately after sending out the message, without blocking or waiting for the response. Since each physical processor may house many migratable objects, the ARTS has a scheduler to decide which object will execute next. This scheduler is message-driven; only objects with a pending message can be chosen to execute.

Other important features of the ARTS include scalable message forwarding for migratable objects and the ability to monitor the CPU usage and communication pattern of the system. Every object has a "home" processor, and when the object migrates, the home processor will keep track of its whereabouts and forward any message to it. If the communication is persistent, the home processor may also inform the sender of the current location of the destination object in order to save the forwarding communication. This distributed forwarding mechanism is scalable and efficient [8], and turns out to be very useful in supporting existing programming models.

The capability of observing CPU and network usage patterns enables several optimizations by

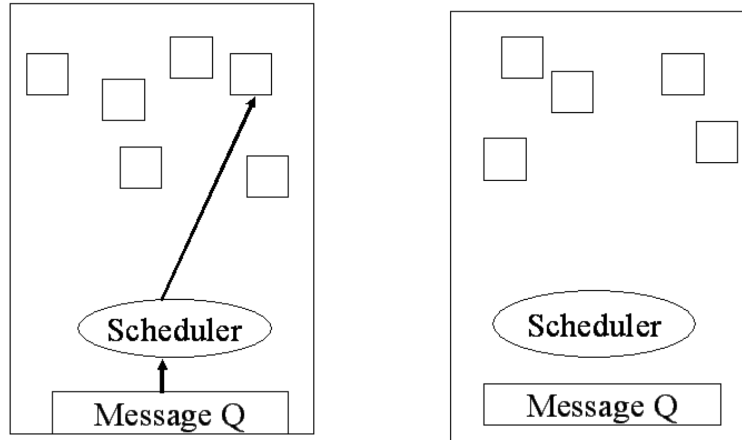


Figure 2.2: Message-Driven Execution With a Processor-Level Scheduler

the ARTS. One of the most important optimizations is automatic load balancing. The workload and communication patterns captured by the run-time are fed into a load balancing strategy chosen by the user, which predicts a more efficient remapping scheme of the objects, and the objects are migrated accordingly. A collection of such strategies was developed by Gengbin Zheng and other group members at PPL through the study of load imbalance in modern applications on supercomputing architectures. A second optimization is of communication. The interconnect usage data is helpful in deciding on the most suitable communication strategy for the current communication pattern, and the run-time is capable of switching the optimization strategy when a change in communication pattern is detected.

In addition to adaptive overlap and automatic load balancing, the ARTS offers system-level support for a collection of features. The migratability of parallel objects naturally allows the objects to migrate to and from hard drives, which enables checkpointing/restart of the program [9]. The ARTS also supports fault tolerance mechanisms based on message logging [10] as well as in-memory checkpointing [11].

A large number of applications have been developed using the Charm++ framework, such as NAMD [12, **?**, 13], a production-level molecular dynamics simulation framework which has demonstrated unprecedented speedups on thousands of processors, and LeanCP [14, 15], a Quantum-Chemistry simulation application. Other examples include rocket simulation [16], computational

cosmology simulations [17], crack propagation simulation [18], space-time meshing with discontinuous Galerkin solvers, dendritic growth in solidification processes, and parallel level-set methods [19].

2.2 Supporting Multi-Paradigm Programming on a Common ARTS

Parallel applications vary in their structures and flows, communication patterns, data organization and access schemes, and so on. Accordingly, different programming paradigms capture the varying nature of the applications. Consequently, next-generation parallel applications composed of multiple components demand support for multiple paradigms for enhancing programming productivity. We propose extending the current ARTS for supporting multi-paradigm programming on a common adaptive run-time system for several productivity benefits. Some of the most important benefits are as follows.

- **Automated Resource Management**

  Developing a high-performance parallel program involves efficiently allocating and managing resources for the program, including processors, memories and network. Achieving efficient resource management is challenging especially for irregularly structured and dynamically varying applications, and consequently such resource management usually demands a significant programming effort. To answer this challenge, Charm++'s ARTS is designed to automate resource management in a parallel environment and significantly reduce the programmer's burden.

- **Concurrent Composibility**

  Concurrent composibility is the ability to automatically interleave the execution of multiple modules in an application such that idle time in one can be overlapped by useful computation in another [20]. Without this ability, the programmer would probably have to break

abstraction boundaries for the sake of performance. Having the shared ARTS among multiple paradigms automates concurrent composibility without losing efficiency or productivity, and also supports co-existence of multiple paradigms in a single application.

- **Common Functions at Run-Time Level**

  A common run-time system can provide common services that are needed across multiple paradigms, such as load balancing, checkpoint/restart support, and communication optimizations. The common run-time system also makes interoperation among different paradigms natural and easy. This capability allows library support across paradigms, further improving programming productivity.

2.3 Approach and Objectives

In this thesis, we explore various directions and techniques encompassing the research topic of *supporting interoperable multi-paradigm programming*. Through our reference implementations, we demonstrate that it is indeed possible to provide support for multi-paradigm programming, and such support does offer productivity benefits to developing complicated and dynamic applications.

Our approach is not to create a single panacea language that handles all paradigms; it is simply not a practical solution to the problem due to the wide variance in the nature of parallel applications. Any general purpose “complete” parallel programming language necessarily becomes complex. Instead, we design several *incomplete* but simple languages/implementations to capture various characteristics of the applications. We offer the programmer the most suitable tools for different paradigms to cover the whole spectrum of large-scale parallel applications of the future, and these incomplete languages can be combined together, thanks to interoperability on the common ARTS.

2.3.1 Thesis Objectives

We observe that many prevalent languages, including MPI and many of the Global Address Space (GAS) languages, cannot separate work-and-data units of different modules due to their processor-oriented model and single execution thread that artificially glues modules together. As an example, in a version of the CSAR codes, the MPI implementation required that the fluids meshes and the solids meshes be glued together on each processor, even though these two different meshes are decomposed separately by each module and thus have no logical (or geometrical) connection. It would improve productivity to have a new paradigm that allows expression of object-level parallelism at the global level for better modularity. Another way to motivate this new language notation is as an enhancement to Charm++ to allow a clear expression of the global view of control in the object-based programming model. Charm++'s object-based model is already able to separate work-and-data units of each module with its own set of migratable objects, leading to better modularity and various performance benefits. In addition, the programming model tends to obscure the global view of control. Because the transfer of control is implemented by asynchronous method invocation among parallel objects, the overall flow of control is fragmented and buried deep in the objects' class code. The first objective of this thesis is **to support the paradigm of describing global view of control in a parallel application**.

Any research on improving productivity in parallel programming must deal with the fact that multiple parallel programming methodologies have evolved over the years. MPI provides an interface for message passing in a processor oriented environment. It has become the most prevalent standard for message passing programming and has been widely used. However, new paradigms of parallel programming have also emerged that compete with and complement MPI. Notable among these are models that support global address space (GAS) of some form, including Global Array (GA) [21], Unified Parallel C (UPC) [22], Co-Array Fortran (CAF) [23], and Multiphase Shared Arrays (MSA) [24]. These different programming models suit different classes of algorithms and applications. The programmer's proficiency and preference can also result in a variety of choice

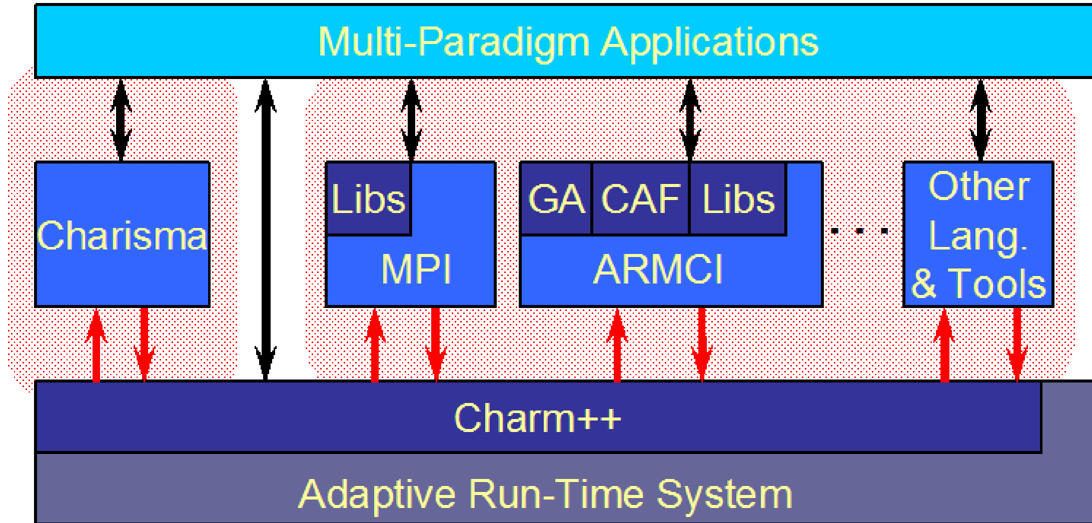


Figure 2.3: Architecture Supporting Multi-Paradigm Parallel Programming on the ARTS

in programming languages and models. Additionally, there are already many parallel libraries and applications written with important existing paradigms. To empower the wider range of parallel models, the second objective of this thesis is **to support adaptivity for these prevalent paradigms on top of a common ARTS to increase interoperability**.

2.3.2 Proposed Architecture

Based on the above objectives, we propose the architecture of adaptivity support for multi-paradigm parallel programming on the ARTS, as illustrated in Figure 2.3. In this architecture, the ARTS serves as the foundation of the programming environment. It provides fundamental functionalities such as abstract communication layer and automated parallel resource management. Immediately on top of the ARTS is the object-based Charm++ language. Charm++, in addition, offers a series of services such as migratable user-level thread support.

Above Charm++ are a number of components which are studied in this thesis. As the first component of the research presented in this thesis, we design Charisma, a high-level language for clear expression of global view of control for parallel programming. Charisma takes advantage of various features of Charm++ and its ARTS to achieve high productivity and performance. It helps

bridge the productivity gap of the message-driven model provided by Charm++.

In the second part, we seek to provide adaptivity support for prevalent paradigms, including MPI, ARMCI, and other languages and environments, on top of Charm++'s ARTS. Our implementations will adapt ordinary code developed with these existing paradigms into "adaptive" programs, so that the ARTS can perform automatic resource management and various other optimizations. Our target is to support a wider variety of applications by adapting them on the ARTS with this approach.

On the whole, we expect to have full interoperability across all these components on top of the common ARTS. Consequently, large scale parallel applications can be built with multiple programming paradigms, and at the same time are able to take advantage of the features and tools of the run-time system.

Chapter 3

Toward a Productive Parallel Programming Language

Although Charm++ has demonstrated its utility in runtime optimizations such as load balancing, and it is more modular than MPI (Refer to [25]), it can be challenging to clearly express the flow of control due to its local view of control, especially for complex applications that involve multiple sets of object arrays. This is demonstrated by the motivating example in next section. Also, in Charm++, methods clearly distinguish the places where data is to be *received*, but the places where data is to be *sent* (invocations) can be buried deeply inside functions of the object code. This asymmetry often makes it hard to see the parallel structure of an application, which is useful for understanding performance issues.

In this Chapter, we first motivate our new language for productive parallel programming through a concrete example and the comparison of several existing parallel programming tools. Then we discuss the design goals for the new language, before we describe our language in next Chapter.

3.1 A Motivating Example

Many scientific and engineering applications have complex structures. Some may involve a large number of components with complicated interactions between them. Others may contain multiple modules, each with complex structures. Unfortunately, for these applications, conventional parallel programming models do not adequately maintain a balance between high performance and

programming productivity. OpenMP [26] programs have a shared view of data and control. The programmer writes code for all the components of the program, with independent loop iterations executed in parallel. This model may be easy to program for a subset of applications, but it is often incapable of taking advantage of large scale parallelism among modules and concurrent control flows, and consequently suffers poor scalability. MPI [27], which represents the message passing model, provides a processor-centric programming model. A parallel job is divided into subtasks according to the number of available processors, and data needed for each subtask is localized onto that processor. Then the user expresses an algorithm in the context of local MPI processes, inserting message passing calls to exchange data with other processes. Basically, it provides a local view of data and a local view of control, although for SPMD programs, the global flow of control is often similar to the local flow of control. Performance wise, MPI programs can achieve high scalability, especially if the program has "regular" patterns, typically with systolic computation-communication super-steps. Some algorithms are simply too difficult to be written in such a fashion. In terms of productivity, this model is fairly easy to program when the application does not involve many modules. Otherwise the programmer will have to first partition the processors between modules, losing the potential performance opportunity of overlapping communication and computation across modules, as well as doing resource management across modules. Some programmers may choose to assign multiple roles to the same group of processors for the sake of performance. With MPI, this results in complexity in writing the message passing procedures, and compromises productivity.

For a concrete example, consider a 3D molecular dynamics simulation application NAMD [28] illustrated in Figure 3.1 (taken from [28]). This simplified version of NAMD contains three types of components. The spatially decomposed cubes, shown by squares with rounded corners, are called *patches*. A patch, which holds the coordinate data for all the atoms in the cube of space corresponding to that patch, is responsible for distributing the coordinates, retrieving forces, and integrating the equations of motion. The forces used by the patches are computed by a variety of *compute* objects, with Angle Computes and Pairwise Computes shown in the figure as examples.

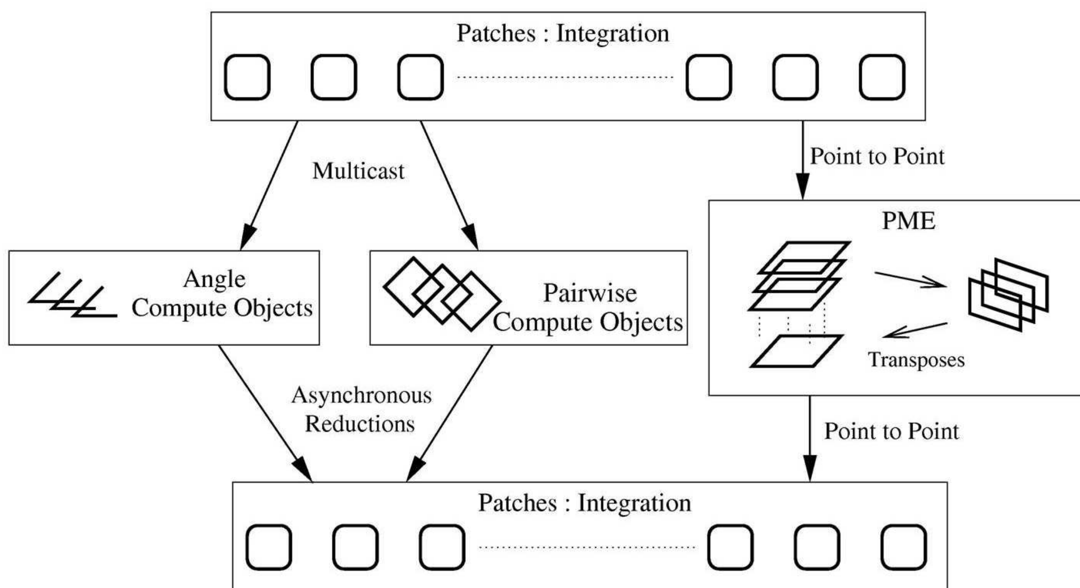


Figure 3.1: Structure of a Molecular Dynamics Simulation Application: NAMD

There are several varieties of compute objects, responsible for computing different types of forces (bond, electrostatic, constraint, etc.). Some compute objects require data from one patch and only calculate interaction between atoms within that single patch. Others are responsible for interactions between atoms distributed among neighboring patches. *PME* objects implement the Particle Mesh Ewald method [29], which is used to compute the long-range electrostatic forces between atoms. PME requires two 3D Fast-Fourier-Transform (FFT) operations. The 3D FFT operations are parallelized through a plane decomposition, where first a 2D FFT is computed on a plane of the grid, followed by a global transpose and a 1D FFT along the third dimension. The simulation in NAMD is iterative. At each time step, the patches send out coordinate data to compute objects and PME objects as necessary, and the compute objects and PME objects perform the force calculations in parallel. Once the resulting force information has been calculated, it is then communicated back to the patches, where integration is performed.

When we consider the various programming models for this relatively simple molecular dynamics application, we often find it difficult to reach a graceful balance between productivity and

performance. Programming with OpenMP, one will write code that, in effect, serializes the phases of coordinate distribution, angle force calculation, pairwise force calculation, PME calculation, force reduction, and patch integration. The flow in such code would look clear, but it is incapable of parallelizing concurrent subtasks, such as angle force calculation and pairwise force calculation, unless wildcard receives with awkward cross-module flow of control are used. Performance and scalability are sacrificed for the ease of programming.

MPI allows the programmer to partition the job into groups of subtasks and assign the subtasks onto partitions of available MPI processes. The programmer can choose to overlap several subtasks onto the same set of processes to keep the CPUs busy. For example, if we have some patch objects and some compute objects residing on the same processor, the patches may use the CPU for the coordinates multicast, and subsequently yield the CPU to the compute objects for force calculation. Since MPI message passing is based on processors, when the programmer wants to express the intention to "send message to subtask S", he/she needs to make the MPI call to send the message explicitly to processor rank K instead of subtask S's ID. Therefore, the programmer has to maintain a mapping between the subtask IDs to the process ranks.

To achieve higher CPU utilization, we want to be able to process the messages as soon as they are received. When the message passing model does in-order message processing with tag matching, the interconnect may deliver out-of-order messages. Therefore, the system overhead of buffering out-of-order arrivals is difficult to avoid. The programmer can take advantage of wildcard source and tag matching, accepting any incoming message, and processing them accordingly. While it is possible to achieve high efficiency, this approach has a major productivity drawback. When there are multiple subtasks from multiple components on one processor, it is difficult to maintain a definite mapping from an arbitrary incoming message to its destination and handler function. The message passing calls will look confusing, and the flow of control cannot be expressed clearly. Modularity is compromised, since to add a new message type to one module (say A), one has to modify code outside the module (in principle, in all modules) to ensure that whenever the message arrives, the appropriate code in module A is invoked.

Charm++, like MPI, provides a local view of control, but unlike MPI, it takes an object-based approach. The programmer writes code for various classes for different subtasks, then instantiates object arrays of arbitrary size from such classes. These objects are assigned onto physical processors by the run-time system automatically, and therefore the programmer does not have to be restricted by the concept of processor. In Charm++'s asynchronous method invocation model, each object's code specifies, in a reactive manner, what the object will do when presented with a particular message. When a message is delivered, its destination object and the method to invoke on that object are stated. Because the message contains information on what to do with it at the receiver side, this can be called an *active message* [30]. Such active messages ensure the prompt processing of data as they become available. In addition, the Adaptive Run-Time System (ARTS) offers further opportunities for performance optimization. However, for complex programs with a large number of object arrays, this comes at a cost of obscuring the overall flow of control: The transfer of control is fragmented by the message sending between objects. To follow the flow of control, one often needs to dig deep into the objects' class code and hop from one to another, and in the meanwhile, to understand parallel operations, such as broadcast, multicast and reduction, among the objects. This poses some difficulty for the expression of the parallel program for both the programmer and its readers.

The above example is not an extremely complicated parallel program. Indeed, it has only three types of components and a few short-running concurrent control flows. A quantum chemistry simulation [15] under development using Charm++ involves 11 different parallel structures, together with complex concurrent flows (See Section 6.1). Clearly, understanding the global control flow is difficult by looking at individual object's codes.

Therefore, the language we design offers an easy mechanism for the programmer to describe the overall view of control: a script-like language notation for orchestrating parallel objects on a global level. In next section, we will take a closer look at this mechanism and the programming pattern it entails.

3.2 Design Goals

In this section, we introduce the design goals of our new productive parallel programming language. The new language offers an efficient parallel programming paradigm to the users, especially those with limited training in parallel programming. Moreover, applying the principles embodied by the new language can be an interesting addition to patterns for parallel programming [31, 32].

In order to justify our design goals, we start with examining the key elements in productive parallel programming in practical scientific and engineering settings. Typical parallel applications are developed in an effort to speed up the solving of scientific and engineering problems. Ideally, the process requires collaboration between two teams: scientists or engineers (referred to as "domain experts" [33]) with knowledge of the problem in its specific domain, and parallel programming specialists with experience in designing efficient parallel flows. The goal is to create a paradigm or pattern through which the two parties can work together to design appropriate tools for the collaboration pattern. In particular, knowledge exchange between the two teams is critical, and a major challenge. There needs to be a common language between the two teams with which the complex ideas can be expressed. Meanwhile, it is also necessary to have certain mechanisms to separate each team's domain so that neither side is distracted by the other team's problems.

3.2.1 Higher Level of Abstraction

During the collaboration described above, it is usually difficult for the two teams to communicate effectively. The obstacle is that they do not speak the same language. While the scientists are enthusiastic about formulas and theorems, the parallel programming specialists think in terms of system details, such as subtask partitioning and message passing. How can they get ideas across in such a scenario? One must develop language mechanisms that allow the domain experts to express their ideas and needs without the technical jargon that is often incomprehensible to the parallel programming specialists. On the other hand, the parallel programming specialists should be able to explain the layout of the parallel flow in a higher level of abstraction.

Therefore, the first design goal of the new language is **a higher level of abstraction in expressing global parallel flow**. The object-oriented approach adopted by Charm++ has been proven to promote modularity and hence is easier to work with, but the overall control flow in a Charm++ program tends to be buried in object code due to the nature of its Actor Model [34, 35]. To overcome this drawback, our language will have the ability to coordinate parallel objects on a global level. Also, a data-driven model of the language can shield the low-level communication details away from the domain experts, making the program easier to understand.

3.2.2 Separation of Parallelism Specification and Sequential Component Development

In the collaborative development process described above, the two teams (application scientists and parallel programmers) have different areas of concern in their respective domains. The second design goal for the new language is a mechanism to separate the two teams' areas of concern. Two key factors in developing a successful parallel program are good sequential performance and efficient parallel flow organization. Good sequential performance can be obtained with a well-designed domain model and highly optimized core computation code. Efficient parallel flow organization depends on the level of parallelism that can be exposed in the program. Success on both these fronts requires the two teams work closely, but it also requires a mechanism to separate each team's domain models from the other. This isolation mechanism improves productivity because it can avoid distraction from the ripple effect caused by the other team's changes. For example, if the domain experts should decide to switch the underlying implementation of a signal transformation algorithm, it should not have a complicated impact on the parallel construct code.

Therefore, it would be ideal if **the sequential part of the code could be kept separate from the parallel constructs**. The separation ensures a clear division of responsibilities. When the domain experts work on the core computation, they do not have to worry about the impact of their code in the parallel context. Typically, the functions will have some input data, do some local

computation, and yield some output data. To the parallel programming specialists, the sequential function can be abstracted to the object's local behavior. Moreover, the task of translating the mathematical formulae and the underlying physics can be restricted mostly to the sequential code, once the basic parallel decomposition has been agreed upon. This further allows the parallel programmers to focus on their area of expertise.

3.2.3 Interoperability with ARTS

The new language we are designing will use new mechanisms to enhance programming productivity. This, however, should not conflict with the goal of retaining existing performance benefits and features from the ARTS. In fact, the third design goal is **to take advantage of the adaptivity benefits with interoperability with ARTS**. Since Charm++'s run-time system already provide a collection of powerful performance optimizations, it is naturally desirable to be able to use them without incurring undue amount of extra programming complexity. The interoperability also gives the program further productivity advantage, because the libraries developed with the new language can be used across platform in other languages such as Charm++ and vice versa.

3.2.4 Proposed Parallel Programming Paradigm

Based on the above arguments, we propose the following paradigm for parallel programming with our new language. The typical development process can be broken down into three steps.

First, the domain experts and the parallel programming specialists work together to design the work/data decomposition and an efficient parallel control flow. At this step, the two teams ignore the detailed behavior of any individual object, and they concentrate solely on constructing a high-level description of parallel control flow with the new orchestration language. With the focused effort, the domain experts can design an overall parallel flow organization while parallel programming specialists can help optimize the parallel flow where necessary.

After the global flow is set, the domain experts can work on their specific core computation.

Now they do not have to be concerned with where the incoming data is from, where the resultant data goes to, or what are the underlying mechanisms of data-driven flow in the program. They simply code the computation in a sequential setting, with given inputs and outputs.

A final process is automatically performed by a translator/compiler of our new language, which translates the orchestration code into remote invocations and integrates the sequential code to generate the final parallel program in Charm++. The translation process fits the sequential components into the big pictures of parallel control flow, and connects them into an efficient parallel program, which can then be further tuned, built and run by the parallel programming specialists.

Last but not least, the above is only an ideal scenario where there is a parallel programming team helping the parallelization process. However, in actual practice, this is often not the case, because many application scientist teams are not blessed with an experienced parallel programming team to collaborate with. When the parallel programming specialists are missing, the domain experts will have to take up the responsibility of understanding and creating the parallel constructs. This reality reinforces the necessity for a higher level of abstraction and more productive development method, which our new language readily supports.

Chapter 4

Charisma: Orchestrating Migratable Parallel Objects

In this Chapter, we introduce our high-level language for orchestrating migratable parallel objects called *Charisma* [36][1]. Charisma extends our efforts toward multiparadigm parallel programming framework as a highly productive language for describing global view of control in complicated parallel applications and libraries.

4.1 Language Design

Charisma employs a macro dataflow approach for productive parallel programming. At the highest level of abstraction, the programmer creates a script-like orchestration program containing statements that produce and consume collections of values. From analyzing such producing and consuming statements, the control flows can be organized, and messages and method invocations can be generated. This idea is similar to the macro dataflow model [37] and the hybrid dataflow architecture model [38], where the data-driven distributed control model is combined with the traditional von Neumann sequential control model. In contrast to the instruction level dataflow models, Charisma's object-level macro dataflow mechanism takes advantage of the message-driven

[1] In historical perspective, the term *Charisma* has been used by a previous project by Milind Bhandarkar. The old Charisma was a common component architecture for parallel programming, and it had the idea of separating sequential code from parallel code. As with our new Charisma, his system also ensured that objects only have sequential data and publish data without having to know the destination. However, it was not focused on the description of global view of control in parallel programming, and did not consider complicated parallel structures and multiple modules.

execution model in Charm++'s and enables dynamic resource management such as automatic load balancing.

A Charisma program consists of two components: the *orchestration code* (in one or more `.or` files) that describes the global view of control, and the *sequential code* (in `.h` and `.C` files) that specifies the local behavior of individual objects. The Charisma compiler generates parallel code from the orchestration statements and integrates the sequential methods to produce the target Charm++ program, which is then executed on the Adaptive RTS. This flow is illustrated in Figure 4.1.

This design corresponds with the design goals of Charisma. First, the script-like orchestration code adopts the macro-dataflow approach, allowing the programmer to express higher level of parallelism abstraction. Secondly, the sequential code is standard C++ code, which facilitates the separate development work or reusing of existing sequential methods by the application scientists. Thirdly, because the resulting program is built and run on the ARTS, it can take full advantage of the performance benefits and features such as adaptive overlap and automatic load balancing.

Since the orchestration code is the center of Charisma notation, we first explain some of the key elements of the orchestration language.

4.1.1 Parallel Object Array

In Charisma, a program is composed of parallel objects. A collection of such objects can be organized into an array to perform a subtask, such as the patches and the force calculators in the previous NAMD example. Although they are called "arrays", these are really a collection of objects indexed by a very general indexing mechanism. In particular, the objects can be organized into 1-D or multi-dimensional arrays that can be sparse, or into collections indexed by arbitrary bit-patterns or bit-strings. One can also dynamically insert and delete elements in an object array. Charm++'s ARTS is responsible for adaptively mapping the object array elements onto available physical processors efficiently.

Moreover, these objects are migratable with support from the underlying ARTS. Once created,

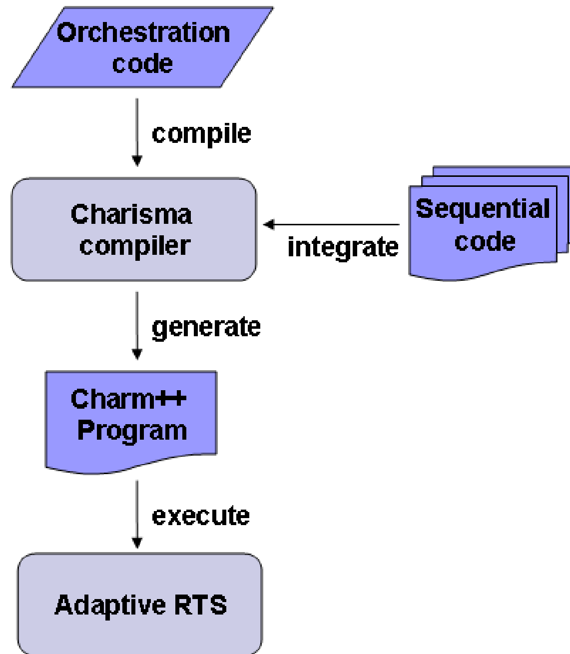


Figure 4.1: Flowchart of Charisma

these parallel objects report the workload at run-time to the system load balancer, and the load balancer will automatically migrate the objects as necessary to achieve higher overall utilization. The message delivery, however, will not be disturbed by the fact that the objects might have been migrated away, because the ARTS implements a scalable message forwarding mechanism [8].

```
class Cell : ChareArray2D;
class CellPair : ChareArray4D;

obj cells :  Cell[N,N];
obj cellpairs :  CellPair[N,N,N,N];
```

Figure 4.2: Charisma Orchestration Code Example: Parallel Object Arrays

In Figure 4.2 is an example of object array declaration in orchestration code for a 2-D Molecular Dynamics (MD) application. The first part is class declaration for class `Cell` and `CellPair`. The second part is the instantiation of two object arrays `cells` and `cellpairs` from these classes. The array `cells` is responsible for holding the atom information in the 2-D partition that corresponds to its index, and the array `cellpairs` does the pair-wise force calculation for a pair of `cells`

objects.

4.1.2 `Foreach` Statement

In the main body of the orchestration code, the programmer describes the interactions between the elements of the object arrays with combinations of orchestration statements. The most common kind of parallelism is the invocation of a method across all elements in an object array. Charisma provides a *foreach* statement for specifying such parallelism. The keywords `foreach` and `end-foreach` form an enclosure within which the parallel invocation is performed. The following code segment invokes the entry method `doWork` on all the elements of array `myWorkers`.

```
foreach i in myWorkers
    myWorkers[i].doWork();
end-foreach
```

Figure 4.3: Charisma Orchestration Code Example: `foreach` Statement

The `foreach` statement looks very much like the `FORALL` statement in HPF [39]. Indeed, they both express the global flow of control. In HPF, `FORALL` provides a parallel mechanism for value assignment of elements of a distributed data array, whereas the `foreach` statement in Charisma specifies the parallelism among the entry method invocation of parallel objects.

The programmer can have multiple statements within one `foreach` enclosure, if those statements are invoked on the same object array with the same indexing. The statements within one `foreach` enclosure cannot be called on different object arrays, because each `foreach` statement is specific to an object array. For the same reason, nested `foreach` statements is meaningless, as a natural consequence of the semantics.

This is really a shorthand notation for having one `foreach` enclosure for each of these statements. Note also that the implementation does not need to broadcast a control message to all objects to implement this. Global control can be compiled into local control, and modulated by data dependences described below.

4.1.3 Producer-Consumer Model

In the MPI model, message passing is specified via the destination processor's rank and communicator, with a tag to be matched. As explained earlier, this mechanism does not always work well in achieving both performance and clear algorithm expression in the presence of complex parallel programs. Charm++'s message delivery specifies the destination object and the function handler. With this information, the destination object knows which function to invoke to process the incoming message. While Charm++ offers a more intuitive way of dealing with communications between subtasks, the programmer still needs to worry about sending and receiving messages while writing an object's local code in a sequential context. To further separate the task of writing communication code for parallelism and composing the sequential computation blocks in a parallel program, Charisma supports producer-consumer communication directly.

In the orchestration code, there is no function call for explicitly sending or receiving message between objects. Instead, each object method invocation can have input and output parameters via *inports* and *outports*. Here is an orchestration statement that exemplifies the syntax for input and output of an object method `workers.foo`.

```
foreach i in workers
    (q[i]) <- workers[i].foo(p[i+1]);
end-foreach
```

Figure 4.4: Charisma Orchestration Code Example: Publish Statement

Here, the entry method `workers[i].foo` produces (or *publishes*, in Charisma terminology) a value `q`, enclosed in a pair of parentheses before the publishing sign "`<-`". Meanwhile, `p` is the value consumed by the entry method. An entry method can have an arbitrary number of published values and consumed values. In addition to basic data types, each of these values can also be an object of arbitrary type. The values published by `A[i]` must have the index `i`, whereas values consumed can have the index `e(i)`, which is an index expression in the form of $\mathtt{i}\pm c$ where c is a constant. Although this example uses different symbols (`p` and `q`) for the input and the output variables, they are allowed to overlap.

The variables that can be used as input and output values constitute the *parameter space* in Charisma. The parameter space resembles the concept of I-Structure and M-Structure [40, 41] in functional languages in their put-take operations and their purpose of exposing parallelism. The variables in the parameter space correspond to global data items or data arrays of a restricted shared-memory abstraction. The programmer uses them solely in the orchestration code to facilitate the producer-consumer model, and has no knowledge of them in the local-view sequential components. A parameter variable can be of an intrinsic or user-defined data type, or a data array, and are declared in the orchestration code as shown below.

```
param error : double;
param atoms : AtomBucket;
param celldata : double [CELLSIZE];
```

Figure 4.5: Charisma Orchestration Code Example: Parameter Variables

In other words, parameter variables appear only in *inports* and *outports* of publish statements. Charisma compiler identifies the inports and outports through parameter variables and uses the ports to connect the statements. Fortran M [42] is similar to Charisma because they both use the concept of *port*. In Fortran-M, ports are connected to create *channels* from which point-to-point communications are generated. It is useful in facilitating data exchange between dissimilar subtasks. Charisma analyzes the *inports* and *outports* of data and generate messages for both point-to-point and collective operations among object arrays, by analyzing data dependences among parameters in the orchestration code. The goal of Charisma is to provide a way of clearly expressing global flow of control in complicated parallel programs. In addition, Charisma is built on top of a powerful adaptive run-time system which offers the generated program performance benefits at no additional cost of programming complexity.

4.1.4 Data Dependence and Program Order

As defined by the language semantics, Charisma uses *program order* to determine data dependence and connect producing and consuming ports. In other words, any consuming statement will look

for the value produced by the immediate preceding statement that publishes the same value in the program order. Any produced value may be consumed by multiple consuming statements. If a producing statement does not have a following consuming statement, the produced value will not have any effect on the program behavior. In a legal orchestration program, any consumed value in any statement should always have its corresponding produced value in a statement. This condition is trivial to satisfy in simple orchestration code which does not contain any loops. We will discuss program order in the presence of loops later in this section.

Beyond the program order restriction of the data flow, Charisma is consistent with Charm++'s asynchronous invocation model. Charisma currently does not support explicit barrier or other synchronization operation supported. The programmer, however, can always enforce a barrier though an artificial reduction operation, either among an object array, or globally among all object arrays.

This also means there is no further implicit barrier between `foreach` statements. For instance, in the code segment in Figure 4.6, `workers[2].bar` waits till the data item `p[1]` is published by `workers[1].foo`, but it does not have to wait after `workers[14].foo` has completed, because there is no implicit barrier between the two `foreach` statements. The execution order is dictated only by data dependence in this code.

```
foreach i in workers
    (p[i]) <- workers[i].foo();
end-foreach
foreach i in workers
    workers[i].bar(p[i-1]);
end-foreach
```

Figure 4.6: Charisma Orchestration Code Example: Program Order

Loops are a frequently used control constructs in parallel application development. They are supported with `for` statement and `while` statement in Charisma, and the rules for data dependence and program order are slightly different from a straight-line program. For the first loop iteration, the first consuming statement within a loop block looks for values produced by the last producing statement before the loop block. For the following iterations, the first consuming statement

matches with the last producing statement within the loop block. At the last iteration, the last produced values will be disseminated to the consuming statement following the loop block.

Take the following code segment as an example, the `coords` produced in the first `foreach` statement is consumed by the first consuming statement in the for-loop. Thereafter, each iteration produces a fresh `coords` from the `integrate` function at the end to be consumed at the next iteration. The produced parameter of `coords` is available after the for-loop, although it is not used here in this example.

```
foreach i,j,k in cells
  (coords[i,j,k]) <- cells[i,j,k].produceCoords();
end-foreach
for iter = 1 to MAX_ITER
  foreach i1,j1,k1,i2,j2,k2 in cellpairs
    (+forces[i1,j1,k1],+forces[i2,j2,k2])
            <- cellpairs[i1,j1,k1,i2,j2,k2].calcForces(
        coords[i1,j1,k1],coords[i2,j2,k2]);
  end-foreach
  foreach i,j,k in cells
    (coords[i,j,k]) <- cells[i,j,k].integrate(forces[i,j,k]);
  end-foreach
end-for
```

Figure 4.7: Charisma Orchestration Code Example: MD Example

4.1.5 Program Determinacy

With the above data dependence semantics, Charisma is designed to be a deterministic language. To ensure that, the implementation needs to satisfy the following deterministic execution constraint. For any individual object, all Charisma methods are always executed in the program order at each run.

Refer to the example in Figure 4.6. It is understood that `workers[2].bar` has to wait for `workers[1].foo` to finish due to data dependence. For the sake of determinacy, however, `workers[2].bar` should also wait for `workers[2].foo` to finish before it can start in order to respect the program order prescribed by the orchestration code.

In straight-line code in Figure 4.6, the implementation uses a state counter in each object to enforce method execution in program order. The counter marks the progress of method invocation and prevents out-of-order execution. In presence of a loop, the counter needs to be reset to point to the beginning of the loop body at each new iteration.

Epoch control is also necessary to enforce deterministic execution of loop statements. Epoch control avoids sending values to the next iteration prematurely. In our implementation, we impose barrier where necessary for epoch control. As future work, we can automate this or implement more intelligent epoch control schemes to achieve higher efficiency.

The following is another illustration of program determinacy in Charisma. In this example, `Si` and `Ri` represents `foreach` statements. If the values produced by any of the `Si` statements are not consumed by any of the `Ri` statements and vice versa, then the two loops can execute in a interleaving fashion without breaking determinacy. Otherwise, the first `while` loop has to complete before the second loop can start.

```
while e
{
    while e1
    {
        S1;
        S2;
    }
    while e2
    {
        R1;
        R2;
    }
}
```

Figure 4.8: Charisma Orchestration Code Example: Program Order with Loops

4.1.6 Describing Communication Patterns

The method invocation statement in the orchestration code specifies its consumed and published values. These actions of consuming and publishing are viewed as input and output ports, and

the Charisma run-time will *connect* these ports by automatically generating efficient messaging between them. Using the language and the extensions described below, the programmer is able to express various communication patterns.

• Point-to-point communication

We now introduce the mechanism to allow point-to-point communication among objects via the producer-consumer model. For example, in the code segment below, `p[i]` is communicated via a message in asynchronous method invocation between elements of object array `A` and `B`.

```
foreach i in A
    (p[i]) <- A[i].f(...);
end-foreach
foreach i in B
    (...) <- B[i].g(p[i]);
end-foreach
```

Figure 4.9: Charisma Orchestration Code Example: Point-to-Point Communication

From this code segment, a point-to-point message will be generated from `A[i]`'s publishing port to `B[i]`'s consuming port. When `A[i]` calls the local function `produce()`, the message is created and sent to the destination `B[i]`. By this mechanism, we avoid using any global data and reduce potential synchronization overhead. For example, in the code segment above, `B[2].g()` does not have to wait on all `A[i].f()` is completed to start its execution; as soon as `A[2].f()` is done and the value p[2] is filled, `B[2].g()` can be invoked. In fact, even before `A[i].f()` completes, `p[i]` can be sent as soon as it is produced, using callback in the implementation.

• Reduction

In Charisma, the publishing statement uses a `+` to mark a reduced parameter whose value is to be obtained by a reduction operation across the object array. Following is an example of a reduction of value `err` on a 2-D object array `A`.

The reduction operation to be used is not specified in the orchestration code. Instead, it is coded in the sequential part (See Section 4.1.7).

The dimensionality of the reduced output parameter must be a subset of that of the array pub-

```
foreach i,j in workers
    (..., +err) <- workers[i,j].bar(...);
end-foreach
...
Main.testError(err);
```

Figure 4.10: Charisma Orchestration Code Example: Reduction

lishing it. Thus reducing from a 2-D object array onto a 1-D parameter value is allowed, and the dimension(s) on which the reduction will be performed on is inferred from comparison of the dimensions of the object array and the reduced parameter.

• **Multicast**

A value produced by a single statement may be consumed by multiple object array elements. For example, in the following code segment, `A[i]` is a 1-D object array, `B[j,k]` is a 2-D object array, and `points` is a 1-D parameter variable. Suppose they all have the same dimensional size N.

```
foreach i in A
    (points[i]) <- A[i].f(...);
end-foreach
foreach k,j in B
    (...) <- B[k,j].g(points[k]);
end-foreach
```

Figure 4.11: Charisma Orchestration Code Example: Multicast

There will be N messages to send each published value to the consuming places. For example, `point[1]` will be multicast to N elements in `B[1,0..N-1]`.

• **Scatter**

A collection of values produced by one object may be split and consumed by multiple object array elements for a scatter operation. Conversely, a collection of values from different objects can be gathered to be consumed by one object. Combining the two, we have the permutation operation.

A wildcard dimension "*" in `A[i].f()`'s output `points` specifies that it will publish multiple data items. At the consuming side, each `B[k,j]` consumes only one point in the data, and there-

```
/* Scatter Example */
foreach i in A
    (points[i,*]) <- A[i].f(...);
end-foreach
foreach k,j in B
    (...) <- B[k,j].g(points[k,j]);
end-foreach
```

Figure 4.12: Charisma Orchestration Code Example: Scatter

fore a scatter communication will be generated from `A` to `B`. For instance, `A[1]` will publish data `points[1,0..N-1]` to be consumed by multiple array objects `B[1,0..N-1]`.

• **Gather**

Similar to the scatter example, if a wildcard dimension "*" is in the consumed parameter and the corresponding published parameter does not have a wildcard dimension, there is a gather operation generated from the publishing statement to the consuming statement. In the following code segment, each `A[i,j]` publishes a data point, then data points from `A[0..N-1,j]` are combined together to for the data to be consumed by `B[j]`.

```
/* Gather Example */
foreach i,j in A
    (points[i,j]) <- A[i,j].f(...);
end-foreach
foreach k in B
    (...) <- B[k].g(points[*,k]);
end-foreach
```

Figure 4.13: Charisma Orchestration Code Example: Gather

• **Permutation Operation**

Combining scatter and reduction operations, we get the permutation operation. Here is an example of 2-D array of values being decomposed and distributed into two 1-D object arrays `A` and `B`, with different orientation of decomposition. `A` holds data along Y-axis and `B` holds data along X-axis. The following code segment is a transposition between the two object arrays.

```
foreach i in A
    (points[i,*]) <- A[i].f(...);
end-foreach
foreach k in B
    (...) <- B[k].g(points[*, k]);
end-foreach
```

Figure 4.14: Charisma Orchestration Code Example: Permutation Operation

4.1.7 Sequential Code

The Charisma programmer supplies sequential code that specifies the local behavior of objects. In a `.h` header file for a specific class, the local member variables and methods that are needed for sequential user code are declared. Note that this header file does not have complete class declaration. It just has the variables and methods declaration used in the sequential code. The definition of those sequential functions is provided in the `.C` files. The `.C` files typically contain function definitions for the class, including those functions that appears in the publish statement in the orchestration code. We will take a closer look that this kind of function with inports and outports.

When composing local functions with consumed and produced values (inports and outports), the programmer does not need the knowledge of the sources of the input data or the destinations of the output data. The input data is seen as parameters passed in, and the output data is published via a local function call. Specifically, for producing, a reserved keyword *outport* is used to mark the parameter name to be produced as appears in the orchestration code, and a *produce* call associates the outport parameter name with an actual local variable whose value is to be sent out. For instance, in the sequential code for `WorkerClass::foo`, the programmer makes a local function call `produce` with *outport* variable `q` to *publish* the value of a local variable `local_q` (assuming `p` and `q` are double precision type).

For reduction, the producing mechanism of connecting a local variable to the outgoing parameter is the similar, only with a different keyword `reduce` and an additional reduction operator/function. The following code segment shows the sequential function `WorkerClass::bar`, in

```
// Sequential function for "(q[i]) <- workers[i].foo(p[i+1])"
WorkerClass::foo(double p[], outport q[]) {
    local_q[i] = ...;
    ...
    produce(q, local_q, n);
}
```

Figure 4.15: Charisma Sequential Code Example: Outport and Producing Values

which a reduction is specified. The programmer calls a local function `reduce` to publish its local value `local_err` and specifies the reduction operation ">" (for MAX). Similar to the `produce` call, an `outport` keyword indicates for which output port parameter this reduce call is publishing data. This call is almost identical to the `produce` primitive, only with an extra parameter for specifying the reduction operation.

```
// Sequential function for "(+err) <- workers[i,j].bar(...)"
WorkerClass::bar(..., outport err) {
    local_err = ...;
    ...
    reduce(err, local_err, ">");
}
```

Figure 4.16: Charisma Sequential Code Example: Reduction

4.2 Library Module Development with Charisma

Charisma is designed to be a language that provides high programming productivity. As reusability is a crucial element in productivity, Charisma must support the ability to develop reusable modules for parallel programs on the ARTS. Due to the difficulty of developing parallel programs and modules, there is a higher premium on reusability of parallel code. Yet, the nature of parallel algorithms makes it harder to design and reuse parallel modules in software development. There are different work partitioning schemes, data distribution schemes, and complicated data flows and control flows to be taken into account. Indeed, the complexity in the interactions between the main program (library caller) and the parallel module (library callee) becomes the most challenging topic

in our research. We first discuss the design alternatives for parallel library interface, and describe the Charisma Module System in detail.

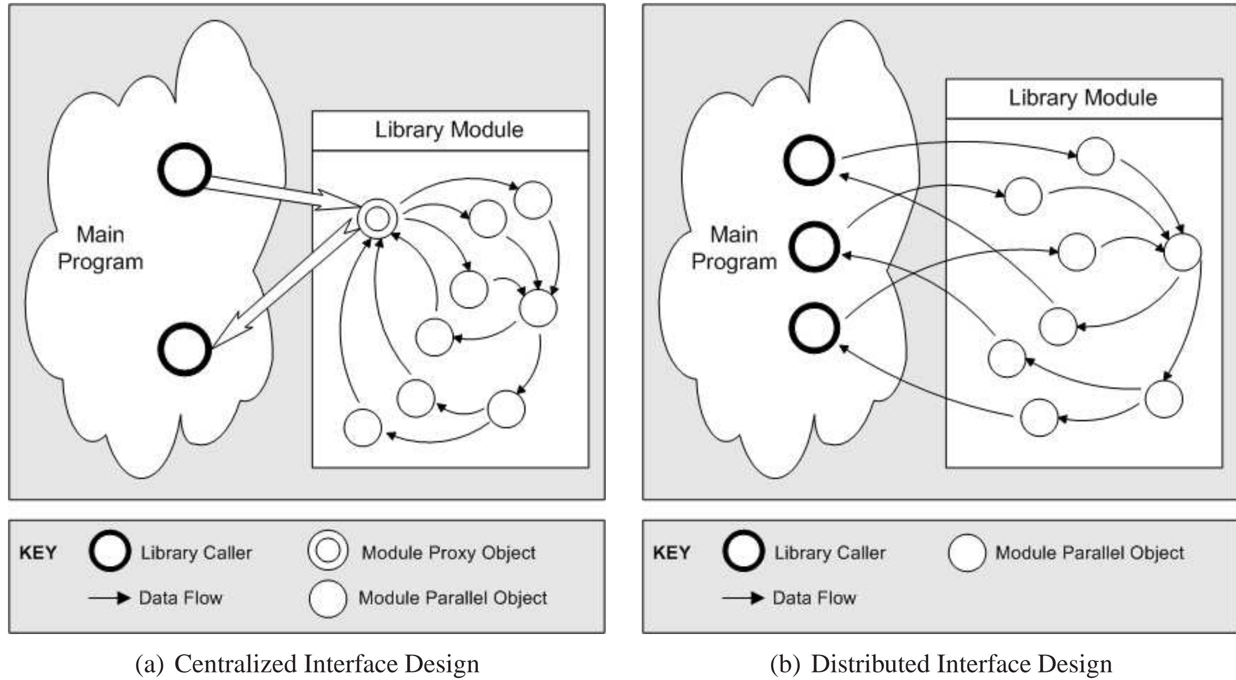
4.2.1 Parallel Library Interfaces

Before discussing parallel library interfaces, we first take a look at how sequential library interact with its caller. Typically, a library module is invoked by calling one or a sequence of subroutines with appropriate parameters. For example, a version of FFTW library [43] requires the caller to call a subroutine to set up the FFT computation, including registering input and output data, a second subroutine to execute the plan, and a final subroutine to destroy the plan. In sequential libraries, the data is passed in and out via subroutine parameters, and the control flow is linear.

For parallel library modules [25], neither the data flow nor the control flow is linear. The data might be fed in and extracted out in a distributed fashion. A synchronization point is not necessarily required before or after the library module is called. There are many ways of invoking a parallel library module and exchanging data with it, and most interactions can be categorized into two patterns: centralized interface and distributed interface, as illustrated in Figure 4.17.

With the centralized interface, the library module exposes a proxy object with subroutines to pass in data and invoke computation. The proxy object may then distributed data into internal parallel objects and initiate the computation in parallel. The results of the parallel subtasks have to be gathered to the proxy object and returned to the caller via a callback. The centralized interaction pattern has better encapsulation of its internals, as only the proxy object's interface is visible to the caller. On the down side, it requires synchronization points and data distribution and gathering that could be eliminated.

In the distributed interaction pattern, the library module and the calling program are more closely coupled. The input data is passed to parallel processes in the library in a distributed and asynchronous fashion, and when the results become available, they are sent back to the callers in the same way. This design improves efficiency by allowing distributed flow of data and avoidance of sequential bottlenecks. The price is weaker encapsulation and added programming complexity.



(a) Centralized Interface Design (b) Distributed Interface Design

Figure 4.17: Interaction Patterns for Parallel Library Modules

For example, the programmer is responsible to ensure the dimensionality of the calling subtasks matches with that of the library processes.

4.2.2 Charisma Module Support

As we discussed above, the distributed interface provides parallel and asynchronous interface which allows closer coupling. However, it could result in higher programming complexity to match up the distributed library interface at the main program side. Fortunately, Charisma has a few features such as parameter space and publish statement that facilitate the interface matching and minimize programmer effort necessary.

Our approach is to integrate the orchestration code from the library into the main program. After the integration, the parallel library caller and callee exist in the same orchestration code and we rely on the existing dependence analysis mechanism to match up the inputs and outputs between the caller and callee.

Because we want to support calling multiple instances of the same library from the same Charisma program, we need to be able to declare and initialize library instances with different names and configurations. Our design requires a library module to post its configuration variables right after its name in the orchestration file, before the library can use any of these configuration variables in the orchestration code or sequential code. Also, the library module declares its inputs and outputs with keyword `inparam` and `outparam`. In the following code segment, the library module `FFT3D` posts 3 configuration variables `CHUNK, M` and `N`, and input parameter `indata` and output parameter `outdata`.

```
// module name with configuration variables
module FFT3D (CHUNK, M, N)
inparam indata;
outparam outdata;

class FFT3DPlanes1 : ChareArray1D;
class FFT3DPlanes2 : ChareArray1D;
obj planes1 : FFT3DPlanes1[M];
obj planes2 : FFT3DPlanes2[M];

param pencildata : complex[CHUNK*CHUNK*N];

// orchestration code that declares the library objects
// and defines their flow with inputs and outputs
begin
    foreach x in planes1
        (pencildata[x,*]) <- planes1[x].fft1d(indata[x]);
    end-foreach
    foreach y in planes2
        (outdata[y]) <- planes2[y].fft2d(pencildata[*,y]);
    end-foreach
end
```

Figure 4.18: Library for Charisma: 3D FFT Example

In the main program's orchestration code, the programmer first includes the library module by name with a reserved keyword `use` followed by the module name. Secondly, the library instances can be initiated with library name and values assigned to its configuration variables. The following example shows how a library instance `fftlib` is created from the module `FFT3D` with a set of

configuration variables.

```
program main-program

// include the module by name
use FFT3D;

// instantiate library instance with parameter initialization
library fftlib : FFT3D(CHUNK=10,M=10,N=100);

// declare classes, objects and params used in main program
class Worker : ChareArray1D;
obj workers : Worker[10];
param fftlib_indata : complex [10*100*100];
param fftlib_outdata : complex [10*100*100];

begin
    // data are consumed and produced by library instances
    // the flows are in parallel whenever possible
    foreach i in workers
        (fftlib_indata[i]) <- workers[i].producePlanes();
    end-foreach
    (fftlib_outdata[*]) <- fftlib:call(fftlib_indata[*]);
    foreach i in workers
        workers[i].getResult(fftlib_outdata[i]);
    end-foreach
end
```

Figure 4.19: Library for Charisma: Using 3D FFT Library

The invocation on a library instance is similar to calling a publish statement, with consumed parameters passed in and published parameters coming out. The only difference is that the function name is in the format of library instance name and the keyword `call` connected with a colon.

With the facility of parameter space, matching the parallel interface becomes easy to code and to understand. Internally, the library instance might have multiple arrays of objects and complex control flows, but they are hidden behind the library interface. In the implementation, the library code is expanded and inserted into the main program with different sets of configuration variables. Then Charisma compiler process the expanded program as one single orchestration file.

In the following code segment, two instances `fftlib1` and `fftlib2` are created, with different sets of values assigned to `FFT3D`'s configuration variables. They each use a set of *inparams*

and *outparams*. The two instances can coexist in the code and their execution can interleave with each other.

```
program main-program

// include the module by name
use FFT3D;

// multiple library instances initialized from same module
library fftlib1 : FFT3D(CHUNK=10,M=10,N=100);
library fftlib2 : FFT3D(CHUNK=4,M=4,N=16);

// declare classes, objects and params used in main program
class Worker : ChareArray1D;
obj workers : Worker[10];
param fftlib1_indata : complex [10*100*100];
param fftlib1_outdata : complex [10*100*100];
param fftlib2_indata : complex [4*4*16];
param fftlib2_outdata : complex [4*4*16];

begin
    foreach i in workers
        (fftlib1_indata[i]) <- workers[i].producePlanes();
    end-foreach
    (fftlib1_outdata[*]) <- fftlib1:call(fftlib1_indata[*]);
    foreach i in workers
        workers[i].getResult(fftlib1_outdata[i]);
    end-foreach
    . . .
    (fftlib2_outdata[*]) <- fftlib2:call(fftlib2_indata[*]);
    . . .
end
```

Figure 4.20: Library for Charisma: Using Multiple Instances of 3D FFT Library

Use of the distributed interface eliminates the sequential bottleneck. For instance, in the above code, `workers[0]` sends `fftlib_indata[0]` directly to the first element in the library module's consuming object array, and that element can start computation immediately upon the receipt of the data. This happens asynchronously with other elements in the same object array, which maximizes the adaptive overlap and improves machine utilization.

4.2.3 Library Support for Charm++

For library support for Charm++, Charisma generates Charm++ code to be incorporated into the main program. Because the generated interface can be too complicated to be human readable, writing code to match the parallel interface is difficult. Therefore, in addition to the distributed interface, we also provide the centralized interface to simplify the interactions between the Charisma library and the calling Charm++ program to hide the complexity.

On the central proxy object (doubled by the library module's main chare), the library module defines 2 function calls `start` and `done`, and the interaction can be unified into a two-phase process.

First, the calling program invokes the library via a `start` call, with input data and callback function. The code segment typically looks as follows. Note that the start function involves two built-in entities in the library's main chare: `doneCB` is the callback object to be sent out at exit of library module, and `next_main_0()` is the function to call to trigger the flow of the library module.

```
void start(char* param, int n, CkCallback& cb){
    // processing of incoming data
    . . .
    // register callback
    doneCB = cb;
    // trigger the flow in the library
    next_main_0();
}
```

Figure 4.21: Library for Charm++: `start`

After the parallel computation, the final results are packed into a message. As the last statement in the library module's generated control flow, the message is sent out with the callback object `doneCB` in a `done` function.

Thanks to the use of asynchronous callback, the efficiency of the library module can be improved by overlapping work from the main program with work in the library. Admittedly, the centralized mode adds extra synchronization that could be eliminated, but the distributed interface

```
void done(void){
    // prepare outgoing results
    msg = . . .
    // send out callback with results
    doneCB.send(msg);
}
```

Figure 4.22: Library for Charm++: `done`

would require the Charm++ programmer understand the details of the code Charisma generated for the library. Since the generated code can be human-unreadable, the distributed alternative is not a good option in this case. In next section, we will explain how a distributed interface is used in library support for Charisma.

4.3 Implementation Issues

After we designed the language, a compiler is implemented to parse code written in the new language and generate Charm++ code. The compiler has the typical components such as lexer, parser and code generator. There are also a few interesting research issues specific to this parallel language. We explain these issues in this Section.

4.3.1 Dependence Analysis

In Charisma's producer-consumer model, the orchestration statements have consumed and published *parameters*. The Charisma compiler collects information of each inport and outport from every statement. The information collected includes parameter name, type, location, subscript of the parameter, and subscript of the associated object array.

Subsequently, dependence analysis [44] is performed to detect if a dependence exists between any inport-outport pair with the same parameter name. The results from the dependence analysis can be organized into a dependence graph among the statements and their parameters. Note that since there will be iterative loop in the control flow, the dependence graph is different from those in the traditional dataflow programs as described in [45]. These dependences represent one or more

messages which need to be generated and pass from the publishing ports to the consuming ports in order to drive the flow of the program.

The dependence testing is capable of identifying loop-carried dependences and loop-independent dependences. When parsing the statements and collecting parameter information, the compiler keeps track of loop level of each occurrence of every parameter. Loop level is useful in creating backward loop-carried dependence and differentiating dependences across loop boundary (needs barrier) and dependences within same loop (does not need barrier).

The Charisma compiler then generates a structures call *triggers* for the dependence graph. If an orchestration statement has an incoming trigger, Charisma will generate code for the corresponding class object to receive and buffer incoming Charm++ messages and drive the subsequent program flow. Likewise, if a statement has an outgoing trigger, Charisma generates code to send out Charm++ messages to the destination. Depending on the communication pattern used, the generated messaging code can range from point-to-point to collective operations such as reduction and multicast/broadcast.

4.3.2 Control Transfer

In the initial version of Charisma, the orchestration statements were numbered and sequenced, and a central main thread was responsible for driving of the control flow. For example, after each parallel `foreach` call, a barrier was imposed, and afterward the main thread decides which statement to invoke next. This clearly was not an optimal implementation. For one thing, the centralized main thread control could cause unnecessary global barriers and synchronization.

In later versions we have remedied this by moving the control transfer code into object code and combining it with a control token issued from the main chare. When an orchestration statement only has data dependence on consumed parameters, it should be fired off as soon as the parameters are published at an earlier statement. An object in an array does not have to wait for other objects in the array to finish their execution of the statement to proceed to the next statement, if the data availability is satisfied. When there is control dependence involved, for instance, for loop-carried

dependence, the main chare will issue a progress token which is needed in addition to the consumed parameters at the triggering of the next statement. This implementation ensures efficient distributed dependence driven control transfer whenever possible.

4.3.3 User Code Integration

Charisma generates Charm++ code that deals with communication and parallel flow of control, from the orchestration code (`.or` file). The programmer supplies sequential code in several separate files: one header file for each class declaring its class member variables and methods, and a collection of C++ files for definition of the class methods. These sequential components will be integrated into the generated parallel code to form a complete Charm++ program, which is compiled and built by the Charm++ compiler.

Charisma offers flexibility for the programmer to integrate other sequential or sometimes parallel code as needed. Additional user code can be included using the `include` keyword in the orchestration code. It can be sequential auxiliary class definitions, or it can be parallel construct that is not automatically generated by Charisma. For instance, in Section 5.1.3, the realtime parallel visualization module is added in the Wator code in the form of included code, because the code is highly specific to the structure of the object array and cannot be automatically generated by Charisma.

4.3.4 Generated Code Optimizations

High productivity programming requires not only fast and easy development process, but also high performance from the output program. We put much effort into optimizing generated code of Charisma. Our methodology is a repeated process of locating the inefficiencies through performance analysis and designing better mechanisms to remove the bottlenecks. Following are two examples of techniques we developed to make Charisma generate better optimized code.

First, we eliminated unnecessary memory copy in the generated code. For a statement that

consumes multiple parameters, because all the values do not arrive together at the same time, an initial design of Charisma buffers the early arrivals in the object's local space, requiring a memory copy. In data-intensive applications such as 3D FFT with an all-to-all transpose operation, our performance visualization shows that the memory copy overhead accounts for 3-5% of total execution time. Then we designed a new scheme that takes advantage of Charm++'s lower level message type, which can hold the data temporarily till all required messages have arrived.

Our second example involves efficient object migration. The efficiency of object migration is affected by the quantity and size of local variables to take along. The most efficient way is to take only those variables absolutely necessary to re-create the object when it arrives on a new processor, but this set of live variables changes with the location where migration is invoked. The foolproof alternative is to copy everything over, to ensure safety of anytime migration. The final version of Charisma is capable of generating code for either case. When the programmer restricts the migration time, the minimal set of live variables are migrated. The programmer can always fall back to the safe mode of migrating all the variables.

4.4 Extensions, Restrictions and Limitations

In this section we discuss the overlap extension of Charisma, as well as restrictions and limitations of the language.

4.4.1 Overlap Extension

Complicated parallel programs usually have concurrent flows of control. If two data-dependence flows are independent in the orchestration code, as in the case of Figure 4.8 when two loops are data independent, they should execute independently and concurrently beyond the restriction of program order. In order to override program order and explicitly express overlapping flows, Charisma provides an extension called `overlap` statement, whereby the programmer can fire multiple overlapping control flows. These flows may contain different number of steps or statements, and their

execution should be independent of one another so that their progress can interleave with arbitrary order and always return correct results.

The code in Figure 4.23 shows an `overlap` statement. The two blocks in curly brackets are explicitly allowed to execute in overlapping flows. Because the program order for data dependence is overridden with the `overlap` statement, in the last `foreach` statement, for example, array `worker2` looks for produced data only from above the `overlap` statement, skipping any outport in the overlapping flow. Their independent executions join back to one at the end mark of `end-overlap` after which program order is resumed.

```
overlap
{
   foreach i in workers1
      (lb[i], rb[i]) <- workers1[i].produceBorders();
   end-foreach
   foreach i in workers1
      workers1[i].compute(lb[i+1], rb[i-1]);
   end-foreach
}
{
   foreach i in workers2
      (lb[i], rb[i]) <- workers2[i].compute(lb[i+1], rb[i-1]);
   end-foreach
}
end-overlap
```

Figure 4.23: Charisma Orchestration Code Example: Overlap Statement on Different Objects

A different `overlap` example is shown in Figure 4.24, where the object array is the same in the overlapping flows, it is natural to raise the question of determinacy. When the `overlap` modifies the program order and allows an object to invoke its methods in arbitrary order, the overall behavior is deterministic if and only if the object states and messages are invariant under all possible interleaving scenarios. In this case, we trust the programmer to ensure that condition for determinacy.

In this example, the user is responsible for guaranteeing that invoking the methods in all possible sequences always give the same outport values and internal object state at the exit of `overlap`

```
overlap
{
   foreach i in workers1
      (lb[i], rb[i]) <- workers1[i].produceBorders();
   end-foreach
   foreach i in workers1
      workers1[i].compute(lb[i+1], rb[i-1]);
   end-foreach
}
{
   foreach i in workers1
      (lb[i], rb[i]) <-
          workers1[i].computeAndProduceBorders(lb[i+1], rb[i-1]);
   end-foreach
}
end-overlap
```

Figure 4.24: Charisma Orchestration Code Example: Overlap Statement on Same Object

block for any element in `workers1`.

4.4.2 Limitations of Charisma

Charisma is designed as a simple language to capture the programming productivity need for a subset of parallel applications. We are confident that Charisma does an excellent job in expressing the global control flow in its targeted applications, which covers a sufficiently big class of parallel programs. With this design, we demonstrate the feasibility of using a simple but restricted language to capture a class of parallel problems. Therefore, the language is not meant to be a complete one that handles every characteristics in parallel applications. When the control flow is data-dependent and determined only at run time, Charisma does not lead to efficient solutions. The issue arises from the difficulty of performing a static capture of dynamic data-driven control flow. The following example illustrates this issue.

Consider a N-Body cosmological simulator called *ParallelGravity* [17] that utilizes the Barnes-Hut tree method [46] to compute gravitational forces. In the tree structure, nodes holding particles are called *TreePieces*, and they are implemented as parallel objects. The index of a TreePiece object is a bit-vector, whose content depends on the depth and location of the node that object

represents in the tree, which is in turn decided by the distribution of the particles in the simulated universe system.

The main difficulty arises from the fact that the dataflow in this simulation is data-dependent. In ParallelGravity, each TreePiece object is responsible for gathering information needed to compute the gravity forces on the particles in that node. According to the Barnes-Hut method, it does not need to collect all the other particles in the universe. Instead, approximation is used whenever reasonable. For TreePieces that are far way, a mathematical approximation is used for an approximate particle-TreePiece force calculation. Only TreePieces that are too close for approximation are fetched and opened up for particle-particle force calculation. Also, to avoid duplicate retrieval of TreePieces, there is a per-processor caching mechanism for remote TreePieces particles.

As a consequence, it is impossible to explicitly specify the dataflow in the simulation before hand. Only at run-time can it be decided which set of far enough TreePiece objects whose centroids can be used for approximation, and which set of nearby TreePiece objects from which full particle information is needed. In addition to this complexity, it is dynamically determined during the tree traversal whether the local per-processor particle cache already has the needed particle information, or new requests are necessary to fetch particle information remote TreePieces.

Due to the incompatibility of such data-dependent dataflow nature and Charisma's dataflow driven method, it is hard to utilize Charisma's facility to lay out the global control flow before hand for this type of application. In this case, we recommend the programmer use lower-level languages such as Charm++ to capture the dynamic flow in the program. Similarly, for algorithms where a global view of data is essential, Multiphase Shared Arrays (MSA) can be used. For its targeted class of applications, as next Chapter will show, Charisma is a power tool to enhance parallel programming productivity without incurring unduly overhead.

4.5 Related Work

Charisma is designed to allow expression of global control flow, and it is not a dataflow language in the traditional sense. In Charisma, objects have persistent states whereas typical dataflow languages are functional. However, Charisma uses macro dataflow to drive the progress of the program and expose parallelism, as most dataflow languages do [45]. In fact, there are a number of parallel programming languages that apply similar ideas.

P-COM^2 [47] is a language with a compiler that composes parallel and distributed programs from independently written components [48]. It proposes a two-phase programming method that separates the individual component development, and the organization and integration of components to form a parallel program. In this sense, it shares the same methodology with Charisma. In comparison, P-COM^2 programs express global control flow implicitly through connecting interfaces of distributed components, whereas Charisma enables explicit description of global control flow with the orchestration language notation.

Many visual parallel languages also adopt the two-step programming methods used by Charisma and P-COM^2. Among them are two interesting examples, HeNCE [49] and CODE [50]. They both treat sequential subroutines as their primitive components, and exploit the expressiveness of visual graph in composing parallel programs. In comparison [51], CODE is more aggressive in its use of dataflow. Each arc in CODE's graph language represent a data movement, and the combination permits expression of various communication patterns. However, the explicit dataflow also increases the complexity of graphs and hence the difficulty of understanding the program. In HeNCE, dataflow is implicit, with the arcs representing control flow, such as invocation of the next components. HeNCE is a better fit for expressing the flow chart of structured programs. Charisma can be thought of combining the benefits of these two approaches, in addition to its own benefits of virtualization. It exposes as much parallelism opportunity as possible with dataflow driven progress, and inserts control constructs where needed to ensure clear expression of the program structure.

Other visual parallel programming environments include VPE [52], a visual programming environment based on PVM [53] that allows explicit message-passing representing tasks with nodes and messages with arcs. In contrast, Charisma is based on object-oriented Charm++. Its underlying adaptive run-time system conveniently provides performance optimizations for parallel programs generated by Charisma.

All these visual parallel programming languages enjoy the advantage of natural expressiveness that is easier to understand. For developers, not all work is done in forms of visual graph. There are typically interface specifications in text, and text annotations in the graph for the nodes and arcs. This complexity is likely to result in a steep learning curve.

Chapter 5

Evaluation of Charisma

We evaluate Charisma by looking at two aspects: performance and productivity. In the first part of our experiments, we compare the performance scalability as well as Source Lines Of Code (SLOC) of a few typical parallel programs written in Charisma and Charm++. In the second part, we show the results from our preliminary productivity study in a classroom setting.

5.1 Performance Evaluation

Now we present three benchmark applications, each implemented with Charm++ and Charisma. We compare the SLOC of both versions and their parallel performance on up to 1024 processors. We use these results to illustrate that Charisma does not incur undue performance overhead while reducing SLOC significantly. The platforms used for the performance evaluation are PSC's Cray XT3 MPP system with 2068 dual 2.6 GHz AMD Opteron compute nodes linked by a custom-designed interconnect, and NCSA's Tungsten Cluster with 1280 dual 3.2 GHz Intel Xeon nodes and Myrinet network.

5.1.1 Stencil Calculation

Our first benchmark is a 2-D 5-point stencil calculation. This is a multiple timestepping calculation involving regions produced by the 2-D decomposition of a 2-D mesh. At each timestep, every region exchanges its boundary data with its immediate neighbors in 4 directions and performs

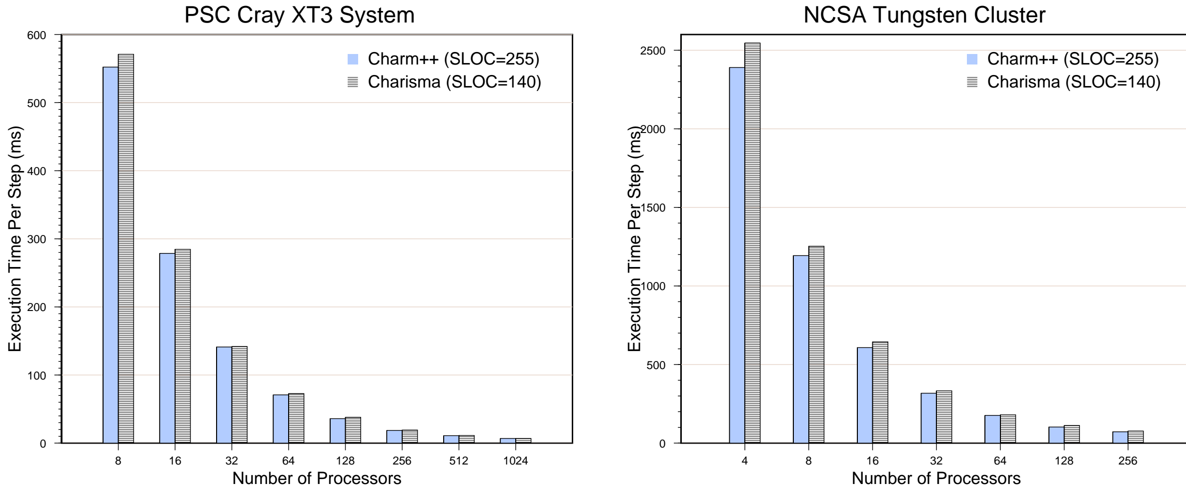


Figure 5.1: Performance of Stencil Calculation

local computation based on the neighbors' data. This is a simplified model of many applications including fluid dynamics and heat dispersion simulation, and therefore it can serve the purpose of demonstration.

Figure 5.1 compares the performances of the stencil calculation benchmark written in Charisma and Charm++. The benchmark problem consists of a 16384^2 mesh decomposed onto 4096 objects. The performance overhead introduced by Charisma is 2 - 6% and the performance scales up to 1024 processors. Because this benchmark is relatively simple, the parallel code in Charm++ forms a significant part of the code. Therefore we see a 45% reduction in SLOC with Charisma.

The overhead can be broken down into two major categories: memory overhead and control overhead. Memory overhead includes overhead incurred by extra data copying and message buffering in the implementation of Charisma. It usually accounts for a larger portion in the total overhead and is dependent on the memory performance of specific computing platform. The rest of the overhead results from parallel control constructs added by Charisma. For example, Charisma program imposes a global barrier at the end of loops to ensure program determinacy. Thanks to efficient implementation of such parallel operations in Charm's ARTS, control overhead has a lower impact on the total performance. Figure **??** illustrates the overhead breakdown of this Jacobi example on

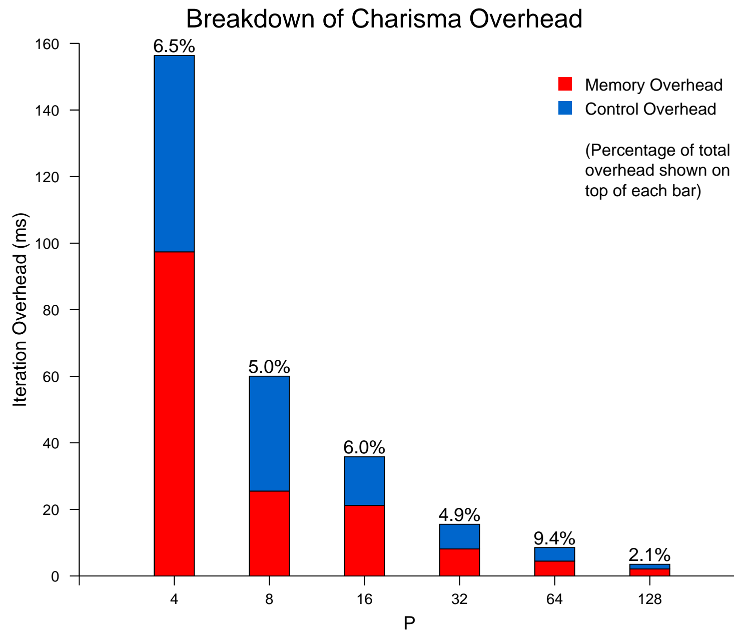


Figure 5.2: Charisma Overhead Breakdownm

the Turing Cluster. The percentage is the total overhead of the total running time for each run.

5.1.2 3D FFT

FFTs are frequently used in engineering and scientific computations. Since highly optimized sequential algorithms are available for 1-D FFTs, multi-dimensional FFTs containing multiple 1-D FFTs on each dimension can be parallelized using a transpose-based approach [54].

We now present the main body of the orchestration code for the transpose-based algorithm for 3D FFT. From this code segment, Charisma generates the transpose operation between the two planes holding the data. Messages are created and delivered accordingly.

Figure 5.4 compares the performance overhead of runs with problem size of 512^3 on 256 objects, on up to 128 processors. From the results, we can see that Charisma, in this benchmark, incurs up to 5% performance overhead, which can be attributed to additional buffer copy for parameter variables. The reduction in SLOC is 37%. In this specific benchmark, sequential components dealing with local 1D and 2D FFT computation constitute a significant portion of the program, and

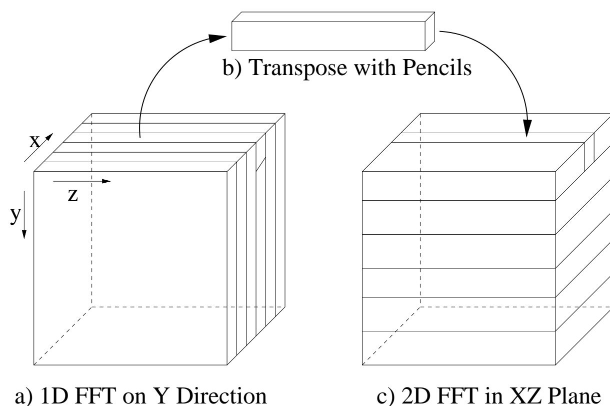


Figure 5.3: Transpose-based 3D FFT Algorithm

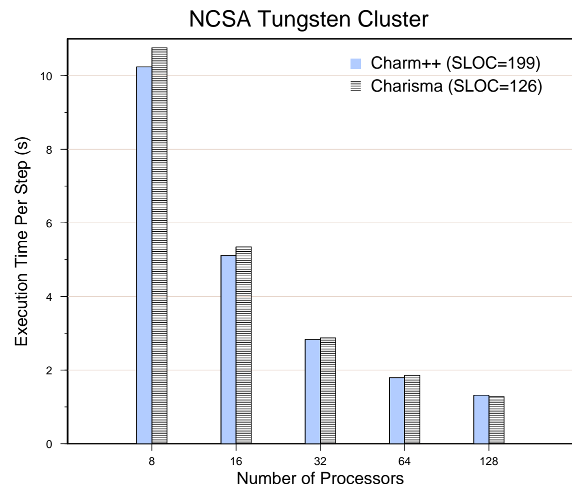


Figure 5.4: Performance of 3D FFT

```
foreach x in planes1
  (pencildata[x,*]) <- planes1[x].fft1d();
end-foreach
foreach y in planes2
  planes2[y].fft2d(pencildata[*,y]);
end-foreach
```

Figure 5.5: Orchestration Code for 3D FFT

therefore the reduction in the SLOC is not as significant as simpler programs. This percentage of SLOC reduction is expected to be even smaller on larger and more complex programs. It must be noted, however, that SLOC alone does not make a good metric of productivity as it does not reflect the actual programming effort. In fact, in more complicated applications, expressing parallel flow of control is far more difficult than in simpler cases, and tools such as Charisma can help programmers code with less effort.

5.1.3 Wator

This program simulates a toroidal water (hence the name "Wator") world. Each cell in the wator world has either a shark or a fish, or water. Sharks and fish interact according to a set of rules. Simple rules describe the movements of sharks and fish and the fact that sharks eat fish. More

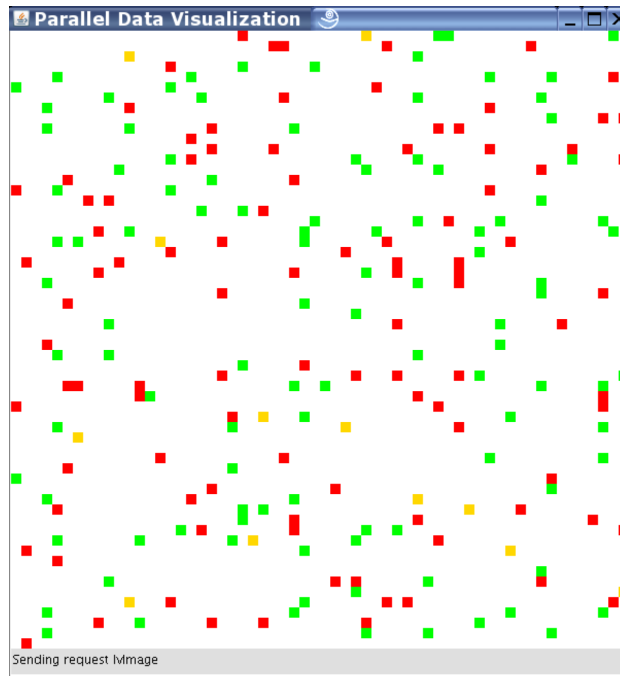


Figure 5.6: Screenshot of Realtime Visualization of Wator

complicated ones may involve breeding, aging and starving of sharks and fish. The program constructs a 2-D decomposition of the 2-D wator world, similar to that in the 2-D stencil calculation. At each time step, objects exchange border information in case sharks and fish move into neighboring cells across an object boundary. Wator is more complicated than a 2-D stencil calculation in that it contains two phases of updating the boundaries: fish action and shark action.

For this experiment, we added two features to the baseline program: automatic load balancing and realtime visualization. Automatic load balancing is a feature provided by Charm++'s adaptive run-time system. Once it is activated, the run-time will monitor the work load of all the processors, and when a load balancing session is triggered, the run-time will migrate objects across processors to achieve a more equal work load distribution. Most of the object migration is done automatically by the system and the programmer only has to provide code for packing and unpacking an object's memory and for triggering load balancing sessions.

The second feature called LiveViz, offers realtime visualization of a parallel program (See Figure 5.6). The GUI client is a standalone Java tool sending periodic requests to the parallel

	Charisma	Charm++	SLOC Reduction
Baseline	253	354	28%
Load Balance	273	383	29%
Visualization	307	407	24%
Both	327	436	25%

Table 5.1: SLOC Comparison of Wator

application and each object answers the request by providing a buffer of color coded values. For instance, in this experiment, different colors are used to represent water, fish and sharks in a cell.

In Table 5.1, we list the SLOC comparison for the baseline program and scenarios with different features added using Charisma and Charm++. A 24-29% reduction in SLOC is observed when Charisma is used to implement these features. Indeed, it is usually easier to add features to a parallel program with Charisma. For example, to add load balancing to the Wator program, a Charm++ programmer needs 29 additional SLOC, while a Charisma programmer needs 20, because some of the code is automatically generated by Charisma.

It is also worth noting that SLOC is a very linear measure. Not all lines of code contribute equally to complexity. In this example as well as other examples where sequential components make up a dominant part of the code, we would get an even larger percentage of SLOC reduction if we separate out the sequential functions and subroutines which are identical in both cases.

5.2 Classroom Productivity Study

Productivity of a HPC language is understood to be more difficult to measure than its performance. In previous work [55, 56, 57], SLOC was used as a major metric of programmer productivity. While SLOC provides valuable information about productivity, it is widely recognized to be an incomplete means of measurement. For this reason, we conducted a preliminary classroom study [58] to further investigate the productivity of programming in Charisma.

(Hours)	2D Stencil		Wator	
	Charm++	Charisma	Charm++	Charisma
Mean	16.84	11.18	24.08	20.47
Median	15.00	10.00	20.00	15.00
StdDev	11.12	6.23	12.39	12.91

Table 5.2: Number of Hours Spent on Development (Sample size 19)

5.2.1 Experiment Environment and Results

Our study involved 25 students in an introductory course in parallel programming. The students included 9 undergraduate students with a CS major, and 16 graduate students from both CS and non-CS majors such as Material Science, Mechanical Engineering, Physics and Aerospace. The average programming experience of the students was 6.16 years, and average years of parallel programming experience was 0.58.

During this course, various parallel algorithms and programming languages/tools were taught and the students were assigned a set of parallel programming tasks, among which were 2D Stencil and Wator, and they were asked to report the time spent (or estimate the time if the task was not actually done) on each programming task with Charm++ and Charisma.

Among the 25 students, 4 did not finish the assignments and hence were unable to provide dependable information. There were 2 entries with numbers too large or too small, so we excluded them as well. The analysis is done on data from 19 students, 13 graduates and 6 undergraduates.

We first looked at the number of hours spent on developing 2D Stencil and Wator programs. The mean, median and standard deviation are listed in Table 5.2. Then we realized that since the students have different levels of experience in programming and different levels of familiarity with parallel programming languages, the percentage of development time reduction is a better metric than the absolute number of hours. Therefore, we calculated the percentage of time saved by using Charisma instead of Charm++, and the mean, median and standard deviation are given in Table 5.3.

Table 5.3 clearly illustrates that the standard deviation for all 19 students is large, pointing to a wide distribution of results. We then tried two different schemes of grouping the students,

(Percentage)	2D Stencil	Wator
Mean	22.35	15.90
Median	26.67	12.13
StdDev	37.76	26.45

Table 5.3: Percentage of Development Time Reduction Using Charisma over Charm++ (Sample size 19)

(Percentage)	Graduate (13)		Undergraduate (6)	
	2D Stencil	Wator	2D Stencil	Wator
Mean	37.84	32.01	-4.21	-10.98
Median	38.33	40	0	-8.33
StdDev	24.38	19.13	43.41	12.56

Table 5.4: Percentage of Development Time Reduction Using Charisma over Charm++ (Graduate vs. Undergraduate)

graduate vs. undergraduate, and CS major vs. non-CS major. The reason behind the grouping is to differentiate between various levels of familiarity with programming and with engineering models. Graduate students usually have more experience with programming in a research context than undergraduates do and the non-CS majors typically have more real-life experience working on scientific or engineering models and have better understanding of the problems that needs parallelization. The results in Table 5.4 and Table 5.5 confirmed the validity of the classification schemes, with the graduate and non-CS groups having much higher average development time reduction and smaller standard deviation in the results, while the undergrad group and CS major group find less merit in Charisma in terms of productivity, with a larger standard deviation.

Another method of visualizing the results is through a x-y spread plot as shown in Figure 5.7.

(Percentage)	Non-CS (8)		CS (11)	
	2D Stencil	Wator	2D Stencil	Wator
Mean	47.14	29.78	4.32	9.44
Median	51.47	37.50	0.00	0.00
StdDev	19.34	15.67	38.15	30.09

Table 5.5: Percentage of Development Time Reduction Using Charisma over Charm++ (CS vs. Non-CS)

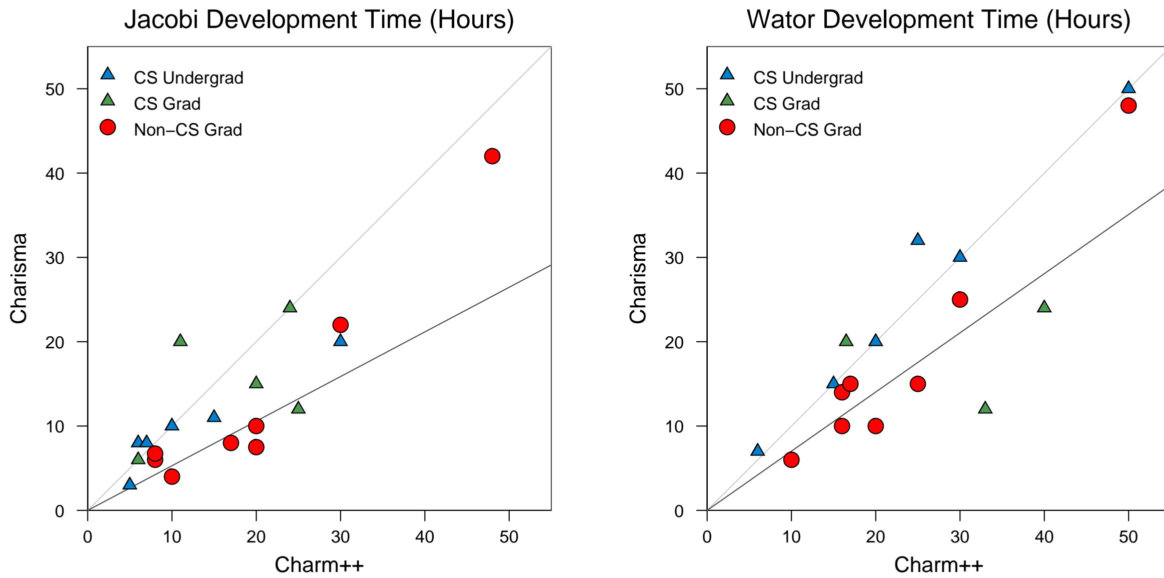


Figure 5.7: Spread Plot of Development Time with Charisma vs. Charm++

The non-CS graduate students (circles), who represent Charisma's target users, are consistently below the 45° line. In contrast, data points for other students (triangles) are spread more widely and on both sides of the 45° line. We discuss the possible reasons in the next section.

5.2.2 Productivity Analysis

The productivity benefit of a parallel programming language notation is decided by a variety of factors. Some factors such as programmer's level of familiarity and experience, and knowledge of problem domains, have little to do with the language itself. Others are properties of the language, like the language complexity. The language complexity has two major components: syntactic complexity and semantic complexity [55].

Syntactic complexity describes the difficulty of transforming an algorithm into source code in that language. Since most programmers are accustomed to the syntax of prevalent programming languages such as C/C++, Fortran, or even Pascal, extraneous syntax beyond that in a new language may make programs harder to write or to read with additional syntactic complexity. Many

parallel programming languages and tools minimally extend existing languages to avoid syntactic complexity. For instance, tools such as UPC and HPF have only simple extensions to the prevalent languages. MPI has language bindings with C/C++ and Fortran, and provides a set of standard library calls to perform the communications. Charisma is also designed with a goal of reducing syntactic complexity by limiting the number of new syntactic features such as publish statement and reusing keywords from existing languages in the orchestration code. In the sequential code, the programmer is asked to write standard C/C++ code. As a future plan, Charisma may incorporation language binding for Fortran too.

A less obvious but arguably more significant issue is semantic complexity of the language. Semantic complexity decides the difficulty of transforming from the sequential problem to the parallel model that fits the language. It is largely independent of the syntax of the language, but more closely connected with the programming model that the language provides, as well as the level of abstraction. For example, MPI programs are written around the concept of processors, and therefore the programmers are expected to take into account the decomposition of the parallel problem into processors, the mapping between processor and subtask, communication optimization, load balancing, and so forth. In contrast, Charisma is built on top of an adaptive run-time system that supports migratable objects and automates parallel resource management. In addition, Charisma allows clear description of the global view of control in a parallel program. All these factors contributes to a reduced semantic complexity when programming with Charisma.

In Section 3.2, we pointed out that Charisma's targeted users are domain experts with ample knowledge in their specific scientific and engineering fields but little parallel programming training. In our classroom study, they are represented by the non-CS graduate students. To these non-CS students, Charisma and Charm++ are equally alien languages to allow a fair comparison on productivity, whereas to CS students, Charm++, being an extension to C++, has some advantage on syntactic familiarity. Furthermore, the non-CS students typically have some experience with programming but not much with parallel programming. When they take the course, they usually have an actual problem in their fields of research that they hope to parallelize. To them, the key

to the productivity of a parallel language is its semantic complexity, or how difficult it is to transform their sequential problems into the parallel model. Charisma's object-based model apparently makes it easier for them to parallelize their engineering problems. To CS students, who do not have much opportunity to work with actual engineering problems, the syntactical factor plays a more important role. They may find the syntactical elements of the orchestration code, such as publish statement and parallel foreach, less than intuitive to use, and are consequently reluctant to accept Charisma.

As this is only a preliminary classroom study, certain aspects can be improved. Firstly, we can extend the set of applications to include examples from more problem domains. Secondly, we can explore a larger metric space. In addition to development time, we can look at time spent on designing, coding, debugging and running. Since what we really care is "time to solution", scaling performance of the resultant parallel program should also be taken into account. Literature on related research on this topic can be found in [59, 60, 61]. We have plans to continue this parallel programming productivity study in the future.

5.3 Code Comparison: MD

In this section, we show how Charisma can overcome some of Charm++'s difficulty of describing global view of control with a concrete example. This example is a simplified version of the NAMD simulation explained in Section 3.1, with only the pairwise force calculation included. `Cells` are the objects that hold the coordinates of atoms in patches, and `cellpairs` are the objects calculating pairwise forces between two `cells`. In the following comparison, definitions for sequential functions such as `Cell::Integrate` and `CellPair::calcForces` are not listed, since they access only local data and should be the same for both versions.

With Charisma, the MD code is listed in Figure 5.8. First, elements in object array `cells` *produce* their coordinates, providing the initial data for the first iteration. During each iteration, `cellpairs` calculate forces by *consuming* the coordinates provided by two `cells` elements. In

```
foreach i,j,k in cells
  (coords[i,j,k]) <- cells[i,j,k].produceCoords();
end-foreach
for iter = 1 to MAX_ITER
  foreach i1,j1,k1,i2,j2,k2 in cellpairs
    (+forces[i1,j1,k1],+forces[i2,j2,k2])
        <- cellpairs[i1,j1,k1,i2,j2,k2].calcForces(
           coords[i1,j1,k1],coords[i2,j2,k2]);
  end-foreach
  foreach i,j,k in cells
    (coords[i,j,k],+energy)
        <- cells[i,j,k].integrate(forces[i,j,k]);
  end-foreach
  MDMain.updateEnergy(energy);
end-for
```

Figure 5.8: MD with Charisma: Clear Expression of Global View of Control

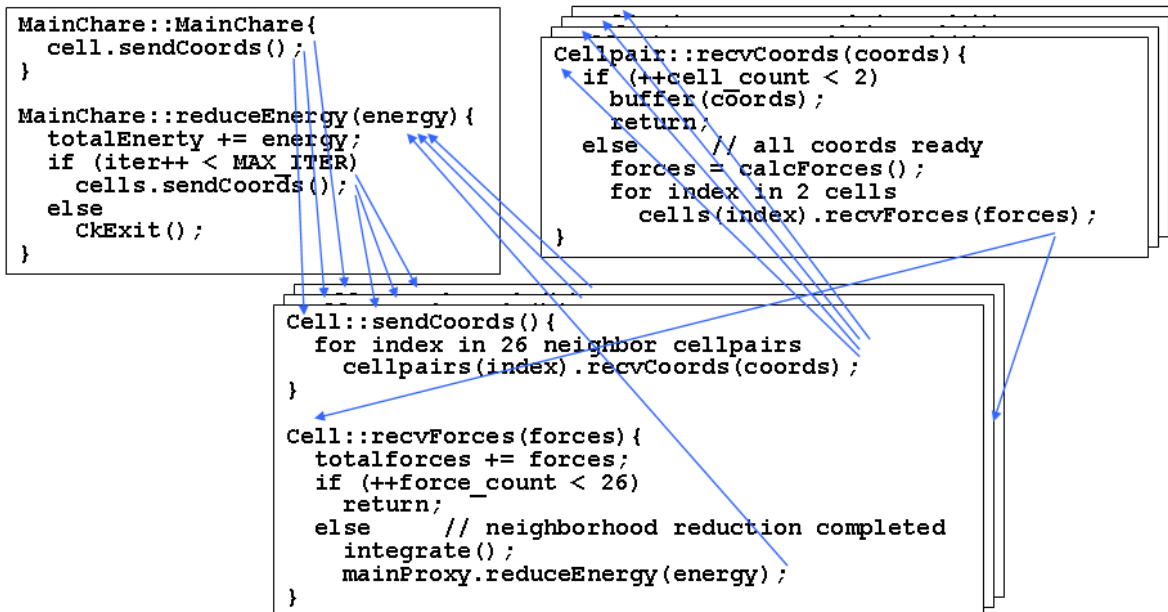


Figure 5.9: MD with Charm++: Overall Control Flow Buried in Objects' Code

```
/* post recvs for all possible messages */
MPI_Irecv(...,reqs[0]);
. . .
MPI_Irecv(...,reqs[K-1]);
/* handle any incoming message
   resulting broken modularity */
while(received<K){
    MPI_Waitany(...,reqs,...);
    switch(GET_TYPE(buf)){
    case (FOR_ANGLE):
        /* calculate angle forces */
    case (FOR_PAIR_LEFT):
        /* calculate pairwise forces */
    case (FOR_PAIR_RIGHT):
        /* calculate pairwise forces */
    }
}
```

Figure 5.10: MD with MPI: Additional Code Required for Performance

the same statement, `cellpairs` *produce* forces combined via a reduction within a `cell`'s neighborhood. These values get *consumed* in the integration phase. The integration also *produces* coordinates for the next iteration and total energy via a reduction operation across all `cells`. In the Charisma code, each orchestration statement specifies which pieces of data it *consumes* and *produces*, without having to know the source and destination of those data items.

Figure 5.9 lists corresponding Charm++ pseudo code for the same program. In three boxes are method definitions for three classes `MainChare`, `Cell`, and `CellPair`, which are typically separated in different C files. Note that `Cell` and `CellPair` are object arrays. To organize the global control flow, one has to dig into the files and hop among them. The control flows, as represented by the arrows, are fragmented and buried in the object code. Following control flow in such a parallel program is more complicated than in sequential object-oriented programming code for two main reasons. Firstly, these arrows do not represent regular function calls as in ordinary object-oriented programs. The results of a function do not return immediately with the exit of that function. Instead, there is a split phase control, where the values are returned with a separate message and method invocation. Secondly, due to the complexity of the parallel operations among

the objects, the control transfers can take form of various communication such as point-to-point communication, reduction, multicast and broadcast. For instance, collecting force data among a `cell`'s neighboring `cellpairs` through a neighborhood reduction requires non-trivial code (not shown in the pseudo code here), and this kind of code is automatically generated in the Charisma version.

The corresponding MPI version will be much more complicated than the Charm++ version. In addition to handling the collective operations, the MPI programmer has to write code for explicitly managing various sets of subtasks, maintaining mapping scheme between subtasks' identities and their physical locations (processor number), and auxiliary code such as load balancing. When the programmer wants to achieve higher degree of overlap between computation and communication, more code is needed to handle the wildcard source and tag matching, as illustrated by the code segment in Figure 5.10.

Chapter 6

Charisma Application Case Study

In this Chapter, we present two applications developed with Charisma. These two applications both represent relatively new methods that become practical with recent development of parallel computing tools. They are both complicated in structure. *LeanCP* has nearly a dozen different parallel objects and overlapping control flows, and topology optimization has multiple rounds of nesting loops in its analysis process. We show the productivity benefits of Charisma through describing the development and results of such applications. The performance results are obtained on the Turing cluster with 640 dual Apple G5 nodes connected with Myrinet network at University of Illinois.

6.1 LeanCP

Many important problems in material science, chemistry, solid-state physics, and biophysics require a modeling approach based on fundamental quantum mechanical principles. The exquisite detail provided by atomistic simulation permits new mechanisms and processes to be easily identified and studied in a control way, and to provide novel insight into well known phenomena that are not understood at a basic level. For example, atom-level simulation study of enzyme structure and reactivity (Figure 6.1) plays an important role in the advancement of science and technology [62].

Among the many variants of atomistic simulation, one important method is molecular dynamics (MD), solving Newton's equations of motion of aggregates of atom and yielding both the structural

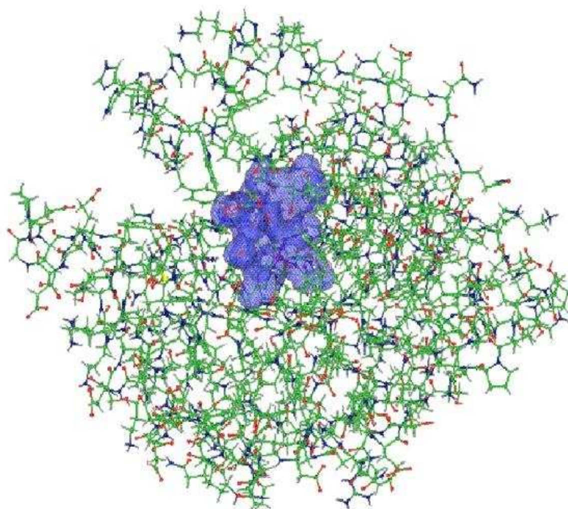


Figure 6.1: Visualization of Human Carbonic Anhydrase. The cloud in the wire frame represents the electron density of the *ab initio* atoms.

and dynamical properties of the simulated system [63]. MD with classical Newton's equations is a powerful tool where the physically motivated gravitational forces are dominant. For more complex systems undergoing reactions, however, an atomistic perspective needs to be taken into account. At this level, the atoms consist of nuclei and electrons, and the system is driven by interactions generated by electrostatic forces between the nuclei and interactions derived from the quantum mechanical solution of electronic energy at fixed nuclear position. While the nuclei remain classical objects subject to Newton's laws, the forces upon them are derived from an *ab initio* or first principles approach [64]. Calculations based on this more advanced *ab initio* approach are more powerful and more computationally intensive.

A particular approach that has been proven to be relatively efficient and useful is Car-Parrinello *ab initio* molecular dynamics (CPAIMD) [65]. Parallelization of this approach is challenging due to the complex dependences among various subcomputations, which lead to complex communication optimization and load balancing problems. In the implementation of CPAIMD with Charm++, called *LeanCP*, the subcomputations are implemented as different sets of object arrays, and the communication optimization and load balancing are handled automatically by the run-time sys-

tem. However, the complicated flow of control is buried deeply inside object code of the various classes. Thus, it makes a perfect example where Charisma can offer higher productivity via clear expression of global view of control while still sustaining the benefits of the adaptive run-time system. It is worth noting that LeanCP is a Charm++ application developed in a collaboration involving a team of scientists and parallel programmers. The collaboration has generated more than 20,000 lines of source code during several years of hard work. As a result, LeanCP code is highly optimized in terms of both parallel programming techniques and scientific algorithms. Since we cannot quite match the effort involved for this study, we aim at making sure that we can capture the overall structure of this application with Charisma.

6.1.1 Implementation with Charisma

In CPAIMD, the ground state electronic energy is calculated by minimizing a functional of the electron density following the principles of Density Functional Theory[66]. The functional contains several terms, the quantum mechanical *kinetic* energy of non-interacting electrons, the Coulomb interaction between electrons or the *Hartree* energy, the correction of the Hartree energy to account for the quantum nature of the electrons or the *exchange-correlation* energy, and the interaction of the electrons with the atoms in the system or the *external* energy. In the last term, the interaction of the valence electrons with the atoms is treated explicitly, and the core electrons are mathematically removed, resulting in what is called the *non-local* energy.

The CPAIMD computation consists of several steps of computation and communication. These steps include (1) computation of the electron density from the electronic states, (2) computation of the exchange correlation and Hartree from the density and the computation of the local electron-particle interaction, (3) computation of the structure factor from the particles, (4) computation of the non-local electron particle energy/forces from the electronic states and the structure factor (5) computation of the lambda-matrix from the forces and (6) computation of the S-matrix from the states.

Figure 6.2 shows the structure of LeanCP, where 11 distinctive object arrays are created to

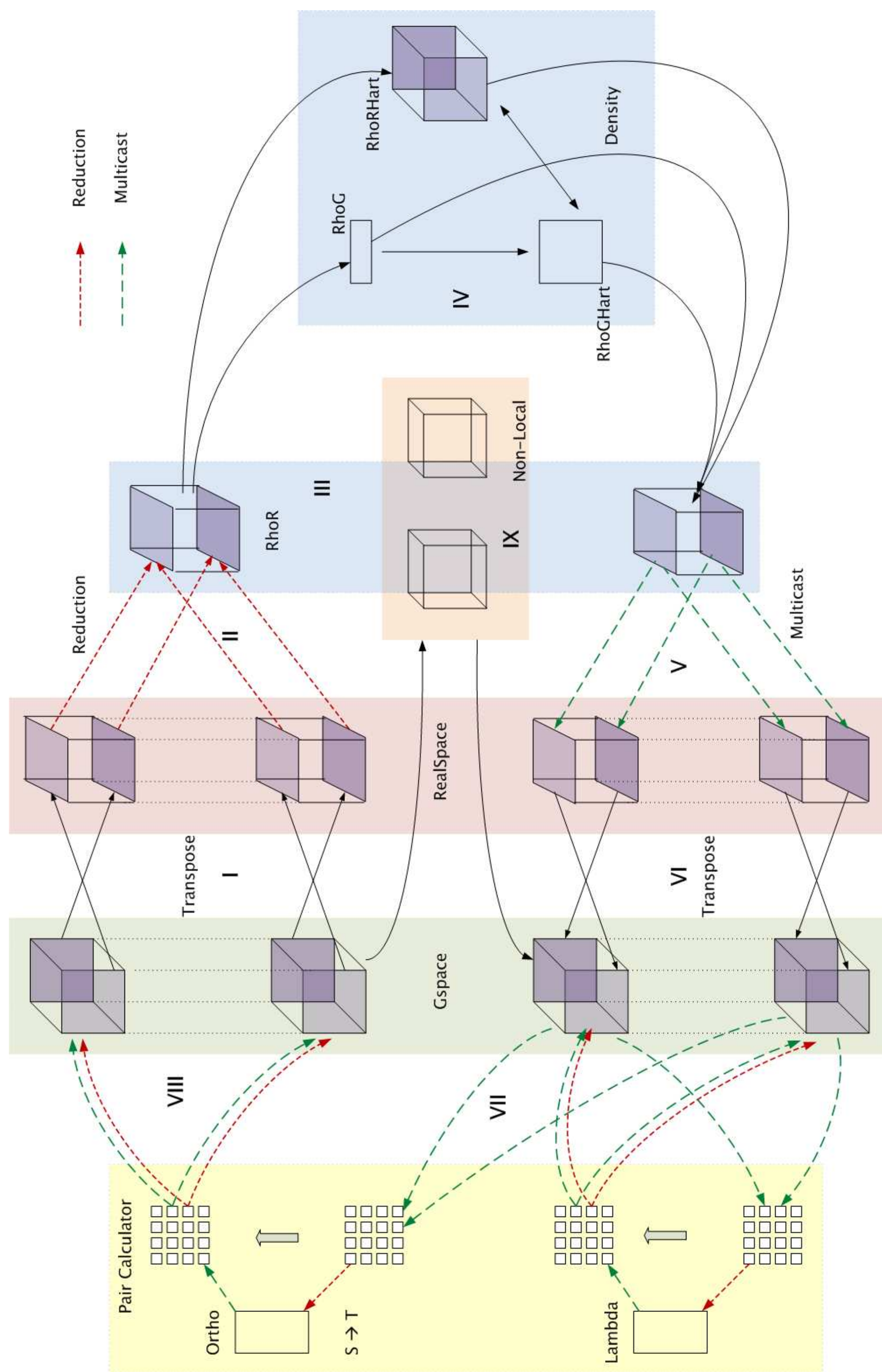


Figure 6.2: Structure of LeanCP

execute different subtasks. The object arrays and their functionalities are listed below.

- **GSpace** and **RealSpace**: Working together, these "State" object arrays create the electronic density in real space (R-Space) and pass it to the "Density" object arrays (Phases I and II). They also receive force data in R-Space from Density object arrays and derive it into G-Space force data (Phases V and VI). The communications between the two object arrays are transpose operations, and the communications between RealSpace and RhoR are a reduction and a multicast. Note that in the figure, boxes representing GSpace and RealSpace arrays are shown twice for Phase I and Phase VI for the sake of clarity.

- **RhoR** and **RhoG**: These two Density object arrays transform electron density from R-Space to G-Space (Phase III), and perform or prepare to perform computation of the "exchange-correlation energy". The exchange-correlation energy has a few components which are derived here with or without the gradient of the electronic density in R-Space (Phase IV). In addition, they copy the Fourier coefficients of the density to the "Hartree" object arrays to compute the "Hartree and external energies". Finally, the energies are sent back to State object arrays with a reduction. The communications involve include transpose operations for 3D FFT, and point-to-point communication between the Hartree objects, and multicast and reduction with the State objects.

- **RhoRHart** and **RhoGHart**: These two Hartree object arrays calculate the Hartree and external energies (Phase IV). Similar to the above Density objects, transpose and point-to-point communications are used.

- **Particle** and **RealParticle**: The kinetic energy of the non-interacting electrons are computed without communication in these two "Particle" object arrays (Phase IX). This flow is independent of the computation in Phases II-VI.

- **Ortho**, **Lambda** and **Pair_Calculator**: After the forces have been computed, a series of matrix multiplications are performed for regularization and ortho-normalization, in these

three object arrays (Phases VII and VIII).

Although the Charm++ implementation could give unprecedented scalability performance [14, 15] on this problem, it suffers from an obscure flow of control, especially at a global view. With the overlapping flows of interactions among the objects shown in Figure 6.2, one can imagine how difficult it would be to follow the control flow in the object-based code at a global level. The sheer number of object arrays and variety of communication patterns involved pose a great productivity challenge to developers and readers of the program alike.

Our goal is to collect first-hand experience in terms of productivity and performance through developing a Charisma version of LeanCP. By examining the orchestration code, we exhibit that Charisma is able to significantly reduce the effort in programming as well as in understanding the flows in the program.

The following code segment is a simple example of a point-to-point communication. When the indexes on the producing object array (`rhoG`) and consuming object array (`rhoGHart`) match with the indexes of the parameter variable `cRGToRGHart`, each object produces and consumes exactly one element in the parameter, and hence the generated communication is a point-to-point operation.

```
foreach y in rhoG
  (...,cRGToRGHart[y]) <- rhoG[y].PhaseIV1(cRRealToRG[y,*]);
end-foreach

foreach y in rhoGHart
  (...) <- rhoGHart[y].PhaseIV1(cRGToRGHart[y]);
end-foreach
```

Figure 6.3: Point-to-Point Operation in LeanCP

The next code segment shows the transpose and reduction operations in Phases II and III. The two object arrays are 2-D: `gSpacePlane` is with $[n_{state}, n_y]$, and `realSpacePlane` is in $[n_{state}, n_z]$. Correspondingly, the parameter variable `cSGToSReal` is 3-D with $[n_{state}, n_y, n_z]$. The wildcard "*" in the produced parameter denotes that each `gSpacePlane` object is producing

multiple (n_z, to be exact) elements of `cSGToSReal` data structure, and the wildcard in the consumed parameter means each `realSpacePlane` collects n_y elements of `cSGToSReal` before the method can be invoked.

```
foreach i,y in gSpacePlane
  (cSGToSReal[i,y,*]) <- gSpacePlane[i,y].PhaseI();
end-foreach

foreach i,z in realSpacePlane
  (+rSRealToRReal[z])
      <- realSpacePlane[i,z].PhaseII(cSGToSReal[i,*,z]);
end-foreach

foreach z in rhoReal
  (...) <- rhoReal[z].PhaseIII1(rSRealToRReal[z]);
end-foreach
```

Figure 6.4: Transpose and Reduction Operations in LeanCP

Similarly, this is the orchestration code for Phases V and VI, involving a multicast followed by a transpose operation.

```
foreach z in rhoReal
  (rSRealToRReal[z]) <- rhoReal[z].PhaseIII2(...);
end-foreach

foreach i,z in realSpacePlane
  (cSRealToSG[i,*,z])
      <- realSpacePlane[i,z].PhaseV(rSRealToRReal[z]);
end-foreach

foreach i,y in gSpacePlane
  gSpacePlane[i,y].PhaseVI(cSRealToSG[i,y,*]);
end-foreach
```

Figure 6.5: Multicast and Transpose Operations in LeanCP

6.1.2 Results

We measured the performance of our Charisma version of LeanCP on the Turing cluster, and the results are shown in Figure 6.6. It is necessary to repeat that we do not expect to match the

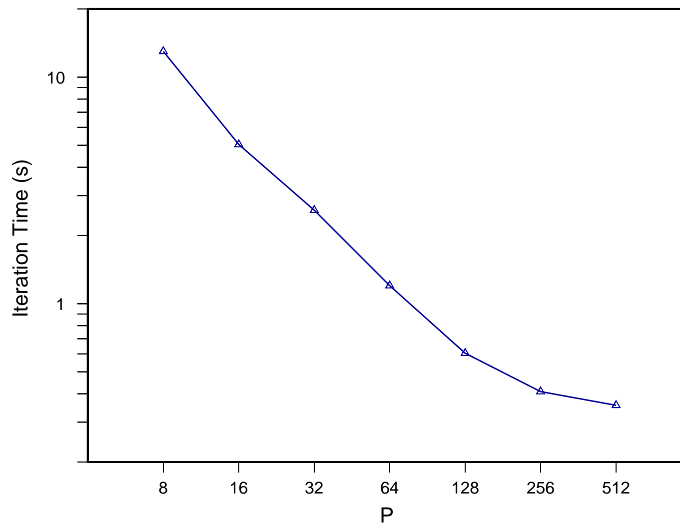


Figure 6.6: Performance of Charisma Version of LeanCP on Turing Cluster

performance of the Charisma version with the original Charm++ version, because it is impossible to repeat the significant amount of effort devoted in the Charm++ version within the given period time allowed for preparing this example. Although our Charisma version is not equipped with many optimizations, it scales smoothly up to 128 PEs.

6.2 Parallel Topology Optimization

The Topology optimization method [67] has become an interesting subject of research with recent developments in structural optimization. Traditional structural optimization aims to find structures such that their size or shape is optimal in a certain sense and satisfies certain constraints. For example, the 3D shape with minimal surface area for a given volume is a sphere. In addition to size and shape, topology optimization takes into account the topology of the structure, including connectivity and boundaries. It has been applied to a wide range of structural design problems in such fields as civil engineering. The topology optimization method is extremely computationally intensive. For this reason, even though the theoretical methods were proposed a long time ago, practical use has not been widespread until recently, as massively parallel supercomputers have

become available [68, 69].

The problem to solve in this specific application is to find a structure that maximizes the transportation of heat from a heat source to the boundary of the design space, given a limited amount of conductive material. It is an interesting problem because the resultant 3D structure will resemble that of blood vessel and capillary system in nature. The heat transfer formulation is elegantly simple, so that the complexity of the finite element analysis can be reduced.

The formal description of the problem is as follows.

$$\min_{\rho_e} \theta(\mathbf{u}, \rho_e) \text{ s.t. } \mathbf{K}(\mathbf{E}_e)\,\mathbf{u} = \mathbf{f} \tag{6.1}$$

$$\mathbf{E}_e = \rho_e^p\,\mathbf{E}_o \tag{6.2}$$

$$\sum_{e=1}^{n} \rho_e v_e \leq V_o \tag{6.3}$$

$$0 \leq \rho_e \leq 1,\; e = 1..n \tag{6.4}$$

where θ is the objective function to be minimized, in this case the temperature at the heat source; n is the number of elements in the domain; E_e is the conductivity corresponding to intermediate densities; E_o is the conductivity of the base material; $K(E_e)$ is the conductivity matrix, which is a function of the material properties E_e; V_o is the upper bound constraint on the total volume of the structure.

6.2.1 Development Process

The process of topology optimization is shown in the flow chart in Figure 6.7. It is an iterative method. First, the initial parameters and conditions are set, and we parameterize the design problem by dividing the domain into finite elements. At each step, the tentative model goes through finite element analysis to obtain the performance of the evolving structure. Then sensitivity analysis is carried out to assess how the performance will change when design variables are changed. The result from sensitivity analysis is used for an appropriate optimization algorithm to update the

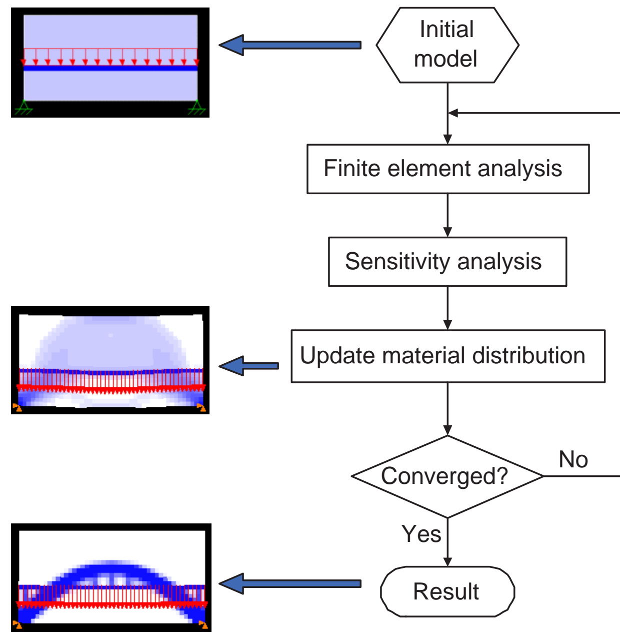


Figure 6.7: Topology Optimization Process

design variables. At the end of the design process, after the evolution of the structure is tested to be converged, each point in the domain has either a point with material or without material (with density of 1 or 0). The optimized structure emerges as material points cluster together to form interconnected members. The material points define the structure in a similar way to how pixels define a image, except that it is in 3D space.

The development team consists of 4 people, with 2 students from a civil engineering background and 2 students from computer science with parallel programming experience. We chose Charisma as the parallel programming tool and follow the development method described in Section 3.2.4. The group was divided into two teams: the *domain team*, which includes the two civil engineering students, is responsible for developing the domain model and preparing sequential components. At the first group meeting, the domain team explained the problem to the *parallel team*, which includes the two computer science students. The parallel team then proposed alternative approaches of partitioning and parallelizing the problem. Thanks to Charisma's higher-level abstraction, it is easy to illustrate how the domain is partitioned into subtasks (parallel objects) and

how they interact during the whole process in a global view. Subsequently, the two team discussed the overheads and complexities of each alternative approach and decided on a final parallelization scheme.

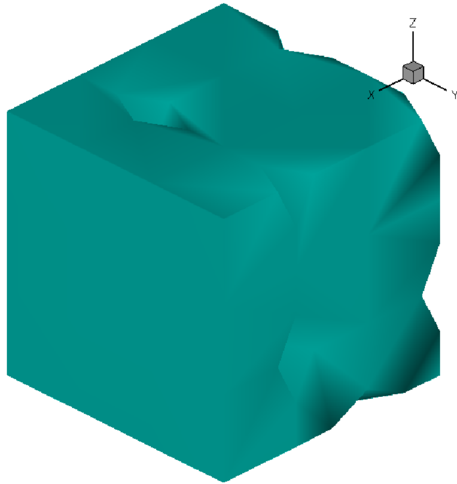
Charisma separates the specification of parallelism from sequential component development, so the two teams can focus on their respective tasks. The parallel team worked on creating data structures used in the parallel data flow (parameter space variables) and organizing the global control flow with the orchestration code. The domain team wrote code (or transformed existing sequential code) for local computations of the parallel objects. With no parallel context involved in the local operations, the functions are simpler for developers on the domain team.

Then the two teams met again to integrate their code. The interfaces of some local functions were slightly adjusted to fit in, and Charisma compiler generated the parallel code which invokes these local computation functions. With the generated parallel code, the domain team was able to verify the correctness with small scale runs, while the parallel team worked on optimizations such as performance tuning and adding load balancing modules, before the program was finally submitted for performance runs.

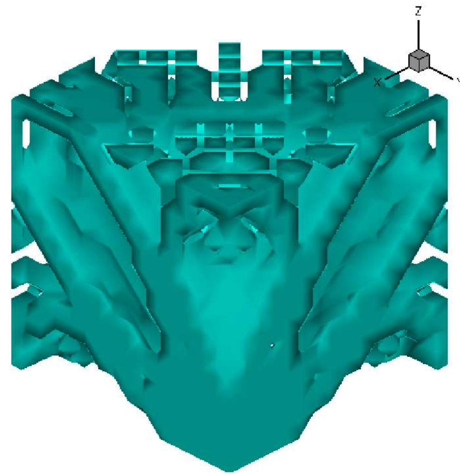
6.2.2 Results

In order to construct high quality 3D structures, a large number of design variables are required, which requires significant amount of computing capacity. Figure 6.8 shows the visualization of the outputs from our application, with different number of elements in the model, and hence different levels of resolution quality as well as different amount of total computation.

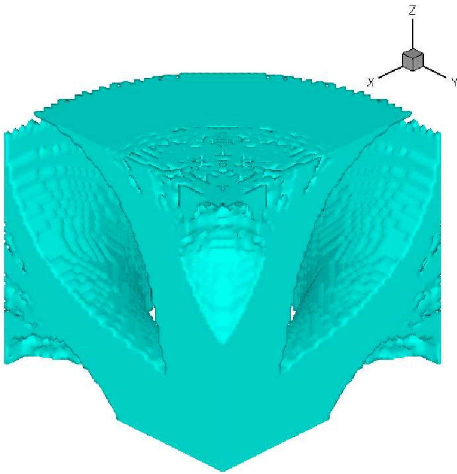
Our topology optimization application with 1,000,000 elements in the mesh scales from 2 to 256 processors on Turing Cluster, as shown in Figure 6.9. The scalability is less than perfect in this case, because the communication between the objects is very heavy, including all the boundary exchange in neighboring faces, edges and corners. These factors are independent of the programming language used. Again, the most interesting part is the development process of this application and how quickly a fairly complicated parallel program can be developed.



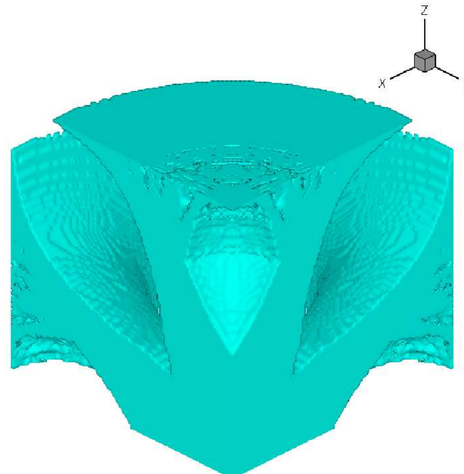
(a) 64 Elements



(b) 4,096 Elements



(c) 262,144 Elements



(d) 1,000,000 Elements

Figure 6.8: Visualization of Optimized Topology for 3D Heat Transfer Problem

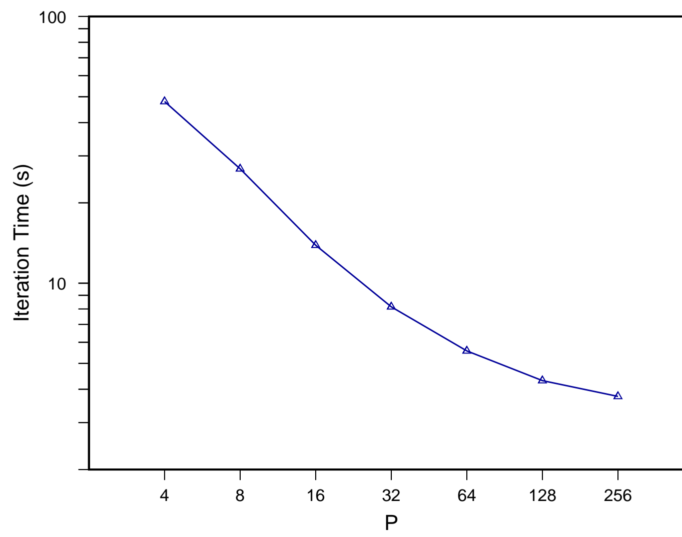


Figure 6.9: Performance of 1,000,000 Element Topology Optimization Application on Turing Cluster

Chapter 7

Adaptivity Support for Prevalent Languages

There are a variety of important programming languages, libraries and tools that offer advanced features and useful programming models. The messaging passing paradigm represented by MPI has established itself as the dominant programming model for scalable scientific applications, with its explicit management of communication. A collection of global address space (GAS) languages compete and complement the message passing model by supporting a global view of distributed data structures. Some of the important GAS languages include Global Array (GA) [21], Unified Parallel C (UPC) [22], and Co-Array Fortran (CAF) [23]. GAS languages furnish high productivity programming abstractions to applications written in standard programming languages such as C, C++ or Fortran.

This thesis proposes providing adaptivity support for important existing parallel programming paradigms, including the message passing and distributed shared memory models, and further supporting existing applications developed with such models and languages. Specifically, we investigate research issues in adaptivity support for Message Passing Interface (MPI) and Aggregate Remote Memory Copy Interface (ARMCI) [70] on the Adaptive Run-Time System (ARTS). MPI and ARMCI represent different data-exchange models, and our research will explore how these models can be developed on the foundation of ARTS.

7.1 Design Goals

Based on the features available on the adaptive run-time system, we set the following design goals for supporting adaptivity in prevalent programming paradigms.

Adaptive overlap between communication and computation: Effective overlap between communication and computation boosts the efficiency of a parallel system and hence is an important performance issue in parallel programming. For example, when an MPI program hits a receive operation, ideally the corresponding message should have already arrived so that the process is not blocked and can avoid wasting CPU time. To achieve this, the programmer tries to move sends up and receives down, and fit some computation between sends and receives, giving time for communication to complete. This manual approach adds to the programming complexity, and even that is often inadequate. Our design should allow adaptive overlap of communication and computation, improving program efficiency without inducing this kind of programming complexity.

Automatic load balancing: Many scientific applications have dynamically varying workload distribution. The computing hotspots can be constantly shifting, as exemplified in the fracture propagation simulation and adaptive mesh refinement methods. For parallel programs load imbalance has an especially high performance impact, because the slowest node dictates the overall performance of the whole system. Run-time load balancing should be effective, adaptive to the application as well as to the parallel platform, and be automated so that minimal user effort is required.

We explore our design alternatives with these goals in mind. We take an over-decomposition approach to adaptive overlapping and virtualize processes with parallel flows of control. Design alternatives for parallel flows of control that we examined include processes, kernel threads, and user-level threads. On a wide range of platforms, we compared various aspects including maximum number of flows per processor, context switch overhead, and migratability [71]. We concluded that user-level thread is the best fit.

Load balancing requires migratability of our user-level threads as well as the capability of the

underlying run-time system to monitor workload distribution dynamically. The run-time should also be able to observe communication patterns happening in the system, so that it can advise communication-aware load balancing strategies. The design should be built on top of a run-time system with such capabilities.

7.2 Processor Virtualization Via Migratable Threads

Our approach to adaptivity support for prevalent paradigms on the ARTS is based on *processor virtualization* [72]. The basic idea is to execute parallel processes with Charm++ migratable objects on each process. Several of these Virtual Processors (VPs) can be mapped onto one physical processor. This gives the run-time system flexibility in resource management for the VPs, and gives the programmer freedom to decompose the parallel job the way that best suits the algorithm.

7.2.1 Charm++ Facilities

Charm++ offers a set of features that facilitate processor virtualization for common programming paradigms. To start with, Charm++'s abstraction of object arrays, which is a collection of objects indexed by any general index structure, turns out to provide a basic functionality needed by most paradigms. The objects are indexed by their rank and the run-time system offers efficient mechanisms for locating objects, redirecting messages to them after migration, and maintaining tables of known locations. Moreover, Charm++'s built-in communication functions, including broadcast and reduction, can also be used for intrinsic communications for some of the paradigms such as `MPI_Bcast` and `MPI_Reduce`.

One of the most important performance benefits of Charm++'s ARTS is automatic load balancing. It is based on an empirical heuristic called *Principle of Persistence* [72], which simply says that for most parallel programs expressed in terms of VPs, the computation loads and communication patterns tend to persist over time. Based on the principle of persistence, our ARTS uses a measurement based load balancing scheme in which a load balance manager constantly monitors

the work load distribution across the system and migrates objects according to a given load balancing strategy to redistribute the work load. The load balancing module is responsible for making decisions and migrating objects around, yet it is the objects' responsibility to ensure the integrity of their data during migration, also called migratability of objects.

Threaded Charm++, or *TCharm*, is a framework built on top of Charm++ that provides common run-time support for migratable and light-weight threads [1]. These threads are created and scheduled by user-level code rather than by the operating system kernel. The advantage of user-level threads are fast context switching [2], control over scheduling, and control over stack allocation. Thus, it is feasible to run a large number of such threads on one physical processor. TCharm threads are scheduled non-preemptively. When another framework needs thread support, the programmer simply "binds" the set of objects from that framework onto a set of TCharm threads. The virtual processes then use the bound threads as needed for blocking and resuming functionalities. The ARTS always migrates the bound objects and threads together.

7.2.2 Implementing Virtual Processes

In order to take advantage of these Charm++ facilities, we choose to implement virtual processes in prevalent paradigms as user-level threads bound to parallel objects, as illustrated in Figure 7.1. The rank of a VP corresponds to the index of the parallel object and is independent of the rank of the physical processor it resides on. As the parallel object and the user-level thread are bound together, they always migrate together, for example, during a load balancing session.

The threads used are TCharm threads. Because they are light-weight user-level threads, we are able to run thousands of threads per processor [73] and keep the context switch overhead at microsecond level. One experiment shows the overhead for suspend/schedule/resume operation is 0.45 microsecond on a 1.8 GHz AMD AthlonXP machine.

[1] TCharm was created by former PPL member Orion Lawlor and maintained by Gengbin Zheng and other group members.

[2] Overhead for a spend/schedule/resume operation is less than 1 microsecond on a 1.8 GHz AMD AthlonXP workstation.

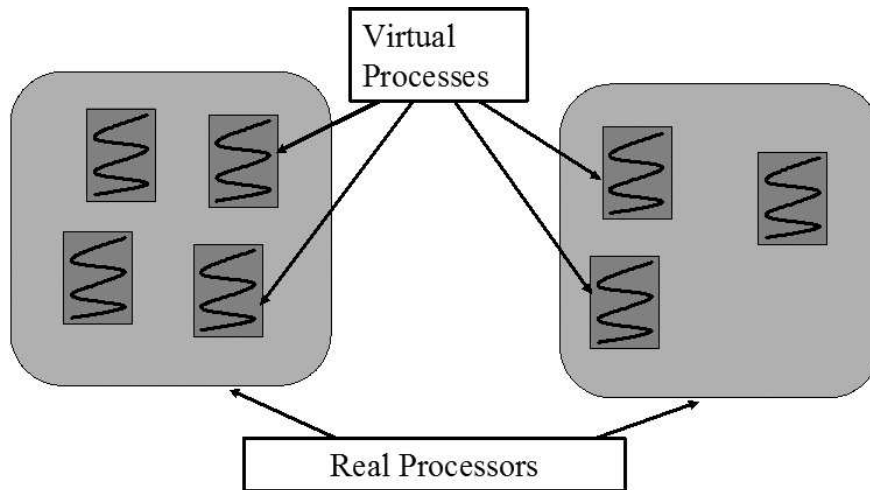


Figure 7.1: Implementation of Virtual Processors

Communication among VPs is implemented as messages among the parallel objects. Although it has been explain in Section 2.1, it is worth reiterating for the sake of clarity how the ARTS supports efficient routing and forwarding of these messages in the presence of object migration. When the destination VP migrates during the transmission of a message, the Charm++ message will be delivered to the object regardless of its current physical location. The ARTS provides a scalable mechanism to determine the location of a given VP. First of all, the system maps any VP index onto a home processor that always knows where the corresponding VP can be reached. When the VP migrates away, it updates its home processor of its current location. A message destined for it will still be sent to its home processor, which in turn forwards the message to its current residing processor. Since this forwarding is inefficient, the home processor will inform the sender of the VP's current location, advising it to send future messages directly to the new location. This mechanism avoids having a central registry which consumes enormous non-distributed storage and presents a serial bottleneck or a broadcast-based mechanism that wastes bandwidth.

7.2.3 Handling Global Variables

The use of global variables can result in a confusing program, since the value of a global variable can be changed by any code in any scope in the program, making the code harder to understand. It can cause potential naming problems when global variables in different modules with the same name cause naming conflict. Despite all this, the practice of using global variables is not uncommon in existing parallel paradigms such as MPI. In our thread-based design, however, global variables can pose an even larger data integrity problem, because when many threads run on one processor, global variables in the user code become unsafe in such a multi-threading environment. Indeed, while a global variable is processor-private in a traditional processor-based paradigm such as MPI, the same "global variable" is meant to be a thread-private variable in our thread-based implementation. To ensure the thread safety of user global variables with minimum programmer involvement, we have explored several alternative approaches.

The first solution is called *swap-global* implemented on run-time level. It is based on the Executable and Linking Format (ELF) in a way similar to the Weaves run-time framework [74]. Previous group member Sameer Kumar together with Gengbin Zheng started implementing this scheme. ELF [75] is a common standard binary file format among various Unix systems including Linux, Solaris, and FreeBSD. In a dynamically linked ELF executable, the set of global variables are accessed via the Global Offset Table (GOT), which contains one pointer to each global variable. The scheme makes a copy of the GOT for each thread, and swaps the pointer to the corresponding GOT when the thread scheduler switches threads. With this approach, the additional context switch overhead is negligible, since it is only swapping a pointer.

On platforms where the GOT does not exists, a variation of this approach makes copies of the global variable data items, and copies in and out the data when threads switch. This alternative, called *copy-global*, is a feasible alternative when the total size of the global variables is not too large.

The third approach is global variable removal done at code level. The idea is to collect all

the global variables into a non-global data structure and pass it into each function referencing any of the global variables. This process is mechanical and sometimes cumbersome. In some simple cases, this task can be performed manually. More than often, the user code is too complicated to remove global variables by hand. Fortunately, it can be automated by source-to-source translation, such as *AMPIzer* [76] based on Polaris [77] that privatizes global variables from arbitrary MPI code in Fortran77 or Fortran90 and generates necessary code for moving the data across processors. Similar tool for handling C/C++ code is also being built based on the source-to-source transformation framework Rose [78, 79] by PPL group members.

7.2.4 Migrating Thread Data

One of the biggest challenges in migrating threads is to migrate thread data. Specifically, it is challenging to extract the useful stack and heap data that belongs to a thread, and to update the correct values of pointers contained in thread data, including function return addresses, frame pointers and pointer variables, on the new processor after migration. Our design includes two different approaches: automatic *isomalloc* and manual *PUPer* functions.

The idea behind isomalloc is to guarantee that a data item of a thread will always have exactly the same address on the new processor as on the old processor. With this guarantee satisfied, no pointers need to be updated because all the references remain valid on the new processor. This idea was originally developed for thread migration in the PM2 run-time system [80].

As illustrated in Figure 7.2, a unused range of virtual address space common across all processors, called *iso-address* area, is reserved on all processors. The iso-address area is then divided into P regions, each for one processor. Note that only virtual memory is reserved, and not any physical memory is actually allocated. When the thread allocates data, the slot corresponding to that thread will be used. When the thread moves to a new processor, the run-time system simply copies over the iso-mallocated data in the corresponding slot, knowing the same virtual addresses are guaranteed to be valid on the destination processor.

While the isomalloc approach enables automatic thread data migration, it requires large virtual

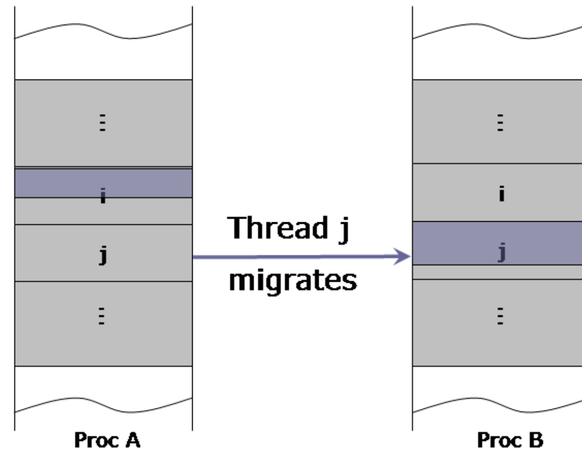


Figure 7.2: Migrating a Thread Stack Allocated with Isomalloc

address space, especially when the number of threads is large. The second solution takes an alternative approach, requiring the programmer to specify in code the data items to move at migration. As an extension to the system's PUP (Packing and UnPacking) framework, the programmer writes a PUP'er function to iterate through the data items chosen to be moved. Compared with the isomalloc approach which copies every piece of data with the thread, this scheme allows the programmer to choose data items to move along. Since the programmer has the best knowledge of the life cycle of the data items, only those essential for restarting the thread execution on the new processor are moved. Therefore, the PUPer solution has an performance advantage at the cost of addition user code.

7.2.5 Automatic Checkpointing

Based on migratable threads, the ARTS supports checkpoint/restart mechanisms with minimal user intervention required. Because the program comprises migratable parallel threads, the ARTS checkpoints the program by migrating all the threads from the processors to stable media: either hard disk drive or memory on peer nodes[11]. At restart phase, the threads are migrated back from the storage, run-time system information and user data restored, and the execution is restarted from where the checkpoint has happened[9].

It is important to note that the checkpoint/restart mechanism in the ARTS has benefits beyond fault tolerance. It also offers the capability of adapting to a changing computing environment. Imagine if we lose 1 node out of a 1024-node partition in the middle of a long execution. We can immediately work around this failure and restart the checkpointed program, with the same number of VPs, on 1023 physical processors. Moreover, this concept can be extended to a shrink/expand feature, which allows an adaptive application executed on the ARTS to shrink or expand the set of physical nodes on which it runs at run-time, adapting to changing load on workstation clusters, or enabling more flexible job scheduling for time-shared machines.

7.3 Adaptive MPI

Adaptive MPI is our implementation of MPI on top of the ARTS. It was started by Milind Bhandarkar [81] and other previous group members with a minimal set of functions implemented as a proof of concept. I continued this project by developing a complete MPI-1.1 implementation and a partial MPI-2 implementation. Together with other group members at PPL, I also carried out a set of performance analysis and applied various performance optimizations. With my work, AMPI has become a mature MPI implementation that are used in real-life applications.

Traditional MPI programs divide the computation onto P processes and typical MPI implementations simply execute each process on one of the P processors. In contrast, an AMPI programmer divides the computation into V virtual MPI processes (VPs). The system maps these VPs onto P physical processors. The number of VPs, V, and the number of physical processors, P, are independent, allowing the programmer to design a more natural expression of the algorithm.

For example, algorithmic considerations often restrict the number of processors to a power of two or a cube number, and with AMPI, V can still be a cube number even though P is prime. When $V = P$, the program executes the same way it would with any typical MPI implementation, and it enjoys only part of the benefits of AMPI, such as collective communication optimization. To take full advantage of the AMPI run-time system, we have V much larger than P.

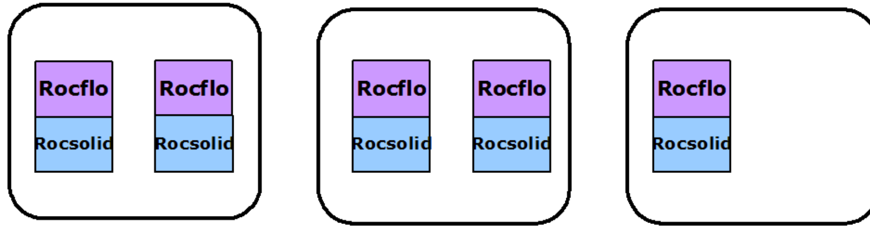


Figure 7.3: Structure of Rocket Simulation Code with Typical MPI Implementation

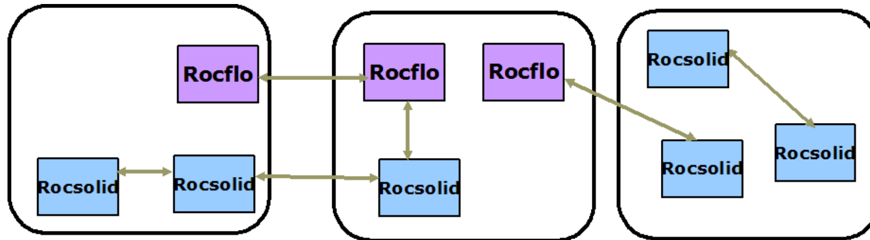


Figure 7.4: Structure of Rocket Simulation Code with AMPI's Adaptivity Support

AMPI offers an effective division of labor between the programmer and the run-time system. The program for each process still has the same syntax as specified in the MPI Standard. Further, not being restricted by the physical processors, the programmer is able to design more flexible partitioning that best fits the nature of the parallel problem. The run-time system, on the other hand, has the opportunity of adaptively mapping and re-mapping the programmer's virtual processors onto the physical machine.

Besides the performance benefits that we are going to present in Chapter 8, this design helps AMPI programmers to practice good software engineering disciplines such as high cohesion and low coupling. High cohesion means any module in a program should be understandable as a meaningful unit and components of a module should be closely related to one another. Low coupling requires that different modules be understandable separately and have low interaction with one another. With MPI's traditional processor-centric programming model, it is often almost inevitable that programmers will violate these principles. With virtualized processes, on the other hand, programmers are given the freedom to partition and structure the parallel application in accordance with good software engineering principles.

For a realistic example, consider a version of the rocket simulation code developed at the Center for Simulation of Advanced Rockets (CSAR) at Illinois [1, 2] (See Section 8.2.1). Figures shown here are simplified representation of the application structure which consists of two modules, *Rocflo* and *Rocsolid*. Rocflo module simulates the structure of fluid dynamics of the burning gas in a rocket booster, and the Rocsolid module models the structure of the solid fuel inside the booster. With typical MPI implementation, the fluid mesh and the solid meshes are required to be glued together on each processor, even though these two different meshes are decomposed separately by each module and have no logical connection, as shown in Figure 7.3. With AMPI's adaptivity support, the work-and-data units of each module with its own set of VPs of different sizes can be separated to allow natural expression of the algorithm, as illustrated in Figure 7.4.

7.3.1 Support for Sequential Replay of an MPI Node

When we use AMPI in our petascale simulation project BigSim at PPL, support for sequential replay of a node in a parallel MPI program is needed for higher simulation accuracy. BigSim [82, 83] is an emulation/simulation system based on the adaptive run-time system built by Gengbin Zheng, Terry Wilmarth and other group members at PPL. Its objective is to allow one to develop, debug and tune/scale/predict the performance of large scale applications for petascale supercomputers before such machines are available, so that the applications can be ready when the machine first becomes operational. It also allows easier "offline" experimentation of parallel performance tuning strategies, without using the full parallel computer. To the machine architects, BigSim provides a method for modeling the impact of architectural choices on actual, full-scale applications. The BigSim system consists of an emulator and a simulator. The emulator can take any Charm++ or AMPI program and run it with a specified number of emulated processors. On the emulator, the application can be tested and debugged with the same number of processes as a performance run, offering a more realistic environment. Also, the emulator can generate traces that are used for timing predictions and performance analysis with the simulator. The trace-driven parallel discrete even simulator is capable of modeling architectural parameters of the target machine.

For more accurate performance prediction, we run the application on a vendor-supplied architectural simulator, where interesting sections of the application can be simulated with instruction-level accuracy. One obstacle is that many architectural simulators operate under a sequential setting only. It is impossible to set up a parallel run of a group of simulators with communication among them. Therefore, support for sequential replay of a node in a parallel AMPI program run is needed.

Our solution is to replicate the change of MPI context on the chosen processor. MPI context includes not only the incoming and outgoing messages, but also MPI environmental variables such as communicators and outstanding requests. Firstly, in communication-related MPI calls, all the incoming messages are logged so that the in-order delivery can be repeated. Outgoing messages are simply discarded since they are not used in the sequential replay anyways. In addition, in MPI environment management calls, such as MPI communicator manipulations, the output parameters are saved in log files. At the sequential replay phase, the application is launched as a stand-alone program. When an MPI call is reached, the output data, such as an incoming message, is read in from the log file, and whenever a non-communication call is reached, the values of its output parameters are retrieved from the log file. With the above support, the sequential replay skips the actual MPI communication on the architectural simulator, allowing it to focus on simulating the more interesting section of the application.

7.4 Adaptive Implementation of ARMCI

Aggregate Remote Memory Copy Interface (ARMCI) is a library for high-performance remote memory copy supported on multiple platforms. It provides an interface for data transfer operations including put, get and accumulate, in both blocking and nonblocking modes. Thanks to its well designed interface, wide portability and low overhead, ARMCI has been used in several global address space languages and parallel distributed-array libraries and compiler run-time systems, including Global Array[21] and Co-Array Fortran Compiler[23].

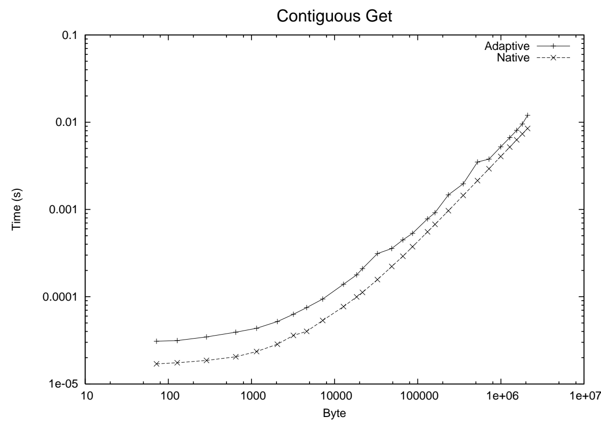
Our effort to support adaptivity in ARMCI started with an preliminary implementation by Chee

Wai Lee and later on developed and maintained by me. Virtual ARMCI processes are implemented with user-level TCharm threads embedded in migratable objects. Each VP encapsulates the state of an ARMCI process required for operations, such as the memory pointers maintained remote copy. As described in Section 7.2.1, each VP is bound to a user-level thread so that they always migrate together during load balancing. To support migratable VPs, memory copy is implemented through messages between objects, because messages can be forwarded by the ARTS to the destination VP even after migration.

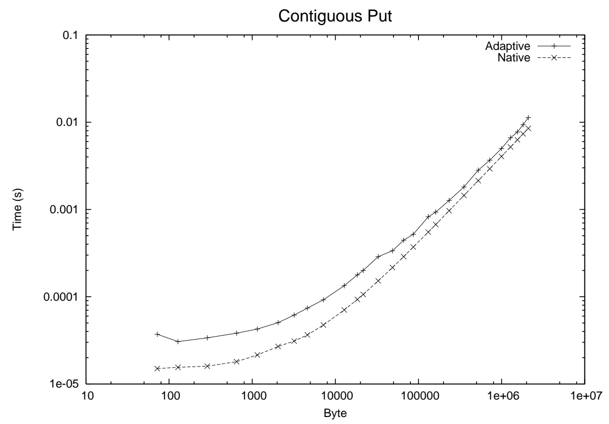
ARMCI provides a collective memory allocation scheme for use with copy operations. The collective call returns to every ARMCI VP an array of pointers to the newly allocated memory on each ARMCI VP. The user then uses these pointers to determine the memory locations for copy operations. In general, to support adaptivity and migration under this scheme for memory allocation, the system would have to broadcast the new pointer locations for each allocated memory block of each migrated ARMCI VP to every other ARMCI VP. We have chosen to implement this memory allocation scheme using *isomalloc heap*, which shares the same idea as in isomalloc stack in Section 7.2.1. When the collective malloc function is called, each VP allocates a region of memory space in its own isomalloc slot for later memory copy use. This ensures a remote address would remain valid after the VP migrates.

7.4.1 Performance Evaluation

In this section we show some preliminary performance results of adaptive implementation of ARMCI. Because our implementation uses messages to implement memory copy, and our current implementation does not make full use of the native RDMA mechanism available, we do not expect to have better performance than the native implementation now. The short message latency of the adaptive implementation is 28μs for get and 27μs for put, about 12μs slower than the native implementation. This is due to an extra ARMCI message header ($\sim$70 bytes) and a 2 - 4 microsecond increase in thread context switch overhead as well as scheduling overhead. For long messages, data in Figures 7.5 and 7.6 show that we pay the overhead of extra message copying in order to

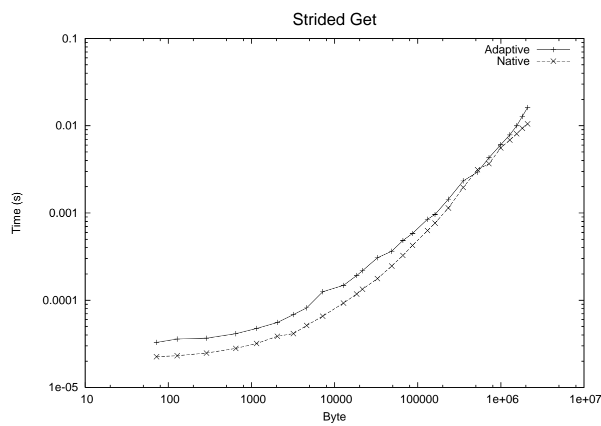


(a) Get

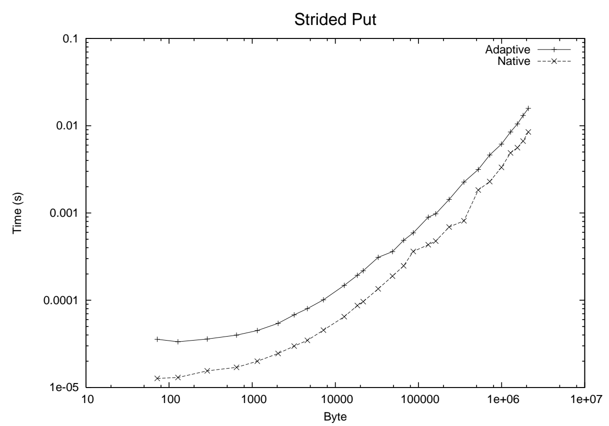


(b) Put

Figure 7.5: Contiguous Copy Performance of Adaptive and Native Implementations



(a) Get



(b) Put

Figure 7.6: Strided Copy Performance of Adaptive and Native Implementations

support migratable objects. It is important to note that adaptivity features such as adaptive overlap and automatic load balancing are expected to enable our implementation to outperform in real-life applications.

7.5 Interoperability Support

Interoperability is essential for large scale parallel application development, and a common adaptive run-time system is an ideal foundation to build the interoperability on. We will describe our support for inter-module and inter-paradigm interoperability in detail.

7.5.1 Inter-Module Interoperability

Large scale parallel applications are usually composed of multiple modules from many disciplines. For example, the rocket simulation application described in Section 7.3 has modules developed by scientists and engineers from various backgrounds, such as fluid dynamics, structural dynamics, and combustion. The modules are developed by different teams in a collaborative way, but each still maintains certain level of independence. In the case of MPI, each module usually has their own `MPI_COMM_WORLD` and their set of communicators. To permit inter-module communication in this kind of application, we have implemented an extension to AMPI. Similar process can be applied to add the extension to adaptive implementations to all other paradigms.

Normally, each AMPI application module runs VPs within its own group of user-level threads distributed over the physical parallel machine. The VPs are organized into communicators such as `MPI_COMM_WORLD` and they start execution from the `MPI_Main` function. When there are more than one modules co-existing within the same executable, there are several things that needs to be changed. First, a top level registration routine needs to be extended to register all the modules' main functions. This tells the run-time system to invoke all the modules from their entrance points. Secondly, a *communicator universe* is added to unite all the `MPI_COMM_WORLD`s so that inter-module communication is possible. A communicator universe, aptly called `MPI_COMM_UNIVERSE`, is im-

plemented as an array of communicators. For instance, in the rocket simulation example, the call to send a message from the Solids VP number 36 to the Fluids VP number 47 looks like the following.

In the Solids module:

```
MPI_Send(nSolids, 1, MPI_INT, tag, 47,
         MPI_Comm_Universe(Fluids_Comm));
```

And in the Fluids module:

```
MPI_Recv(nSolids, 1, MPI_INT, tag, 36,
         MPI_Comm_Universe(Solids_Comm), &stat);
```

7.5.2 Inter-Paradigm Interoperability

Beyond inter-module interoperability within the same paradigm, it is equally if not more important to enable inter-paradigm interoperability. In our context, it is the capability of interoperate between a Charm++ module and an AMPI or ARMCI module on the common ARTS. In fact, our adaptivity support enables one to run legacy code in these prevalent paradigms on the adaptive runtime system, with the VPs executed as Charm++ objects. This makes the interoperation between modules across paradigms a natural and easy process. For instance, in the MPI or ARMCI code, the programmer can create Charm object arrays and invoke entry methods on them. In a Charm++ or Charisma program, the user can also take advantage of any function calls in MPI or ARMCI.

The following code segment is an example of creating a Charm++ object array `myArrayID` in an AMPI program. VP 0 initializes the object array, broadcasts its array ID to all VPs, where the array elements are locally inserted.

Inter-paradigm interoperability makes library support across paradigms possible. We can reuse a library developed with Charm++ in an AMPI program, and vice versa. In real life, we have been using a Charm++ library called Multiphase Shared Arrays (MSA) [24] in an AMPI-based framework, Parallel Framework for Unstructured Meshing (ParFUM) [84]. MSA provides a global data abstraction with a migratable Charm++ object array holding the data pages, and it enforces a

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
if (myRank == 0){
  //array binds to TCharm array
  CkArrayOptions opts;
  opts.bindTo(TCharm::get()->getProxy());
  myArrayID = CProxy_UserArray::ckNew(opts);
}

// myArrayID is broadcast to every node
MPI_Bcast((void *)&myArrayID, sizeof(CkArrayID),
          MPI_CHAR, 0, MPI_COMM_WORLD);

// Array element is locally inserted
myArrayID(myrank).insert();
```

Figure 7.7: Example of Creating Charm++ Objects in an AMPI Program

disciplined multiphase access mode. The three available modes include multiple-read mode, one-write mode, and accumulate mode. Used in ParFUM, MSA is a global hashtable to store elements on shared edges. Partitions in the ParFUM framework contribute elements on a particular edge in MSA's accumulate mode, and read elements on a shared edge in multiple-read mode.

7.6 Related Work

Several projects have put significant efforts into providing benefits of adaptive overlap between computation and communication and automatic load balancing. Because of MPI's popularity among parallel programmers, most of the related projects we discuss here are based on MPI model.

Some MPI implementations support multi-threading programming within one processor to expose an additional degree of concurrency and exploit overlapping between communication and computation. For instance, TMPI [85] uses a two-level multi-threaded design to optimize the scheduling process. TMPI focuses their research for shared memory SMP machines on minimizing CPU spinning and exploiting cache affinity. AMPI, in contrast, optimizes execution for all kinds of supercomputing platforms, even heterogeneous grids [86, 87].

Mobile MPI [88] shares the same over-decomposition strategy as AMPI and uses threads to

execute MPI tasks too. It supports running MPI programs across a heterogeneous environment like grid, but it does it through a portable checkpointing system, which requires a checkpoint/restart process. AMPI does not have this limitation, because it is based on truly migratable user-level threads.

As an extension to MPI, hybrid programming model with the combination of MPI/OpenMP [89, 90] approaches the problem by redistributing OpenMP threads among MPI processes to expose more parallelism, but this approach entails significant programming complexity overhead and requires rewriting existing MPI code, which can be prohibitively difficult for legacy code base.

To support automatic load balancing in traditional MPI model, there are two different approaches. The first is run-time process migration support. Early efforts in migrating MPI process include those in the CoCheck [91] and related Tool-Set project [92]. Since traditional MPI map parallel tasks onto OS processes, the migration process incurs expensive overheads. In contrast, AMPI's migratable threads minimize the load balancing overhead. Additionally, because AMPI typically maps many VPs per physical processor, load balancing is more natural and effective when VPs are moved instead of processes.

The alternative approach to dynamic load balancing is through library support. It improves performance for a specific category of parallel applications. For example, Zoltan [93] is a programming toolkit that supports dynamic load balancing, but its interface is incompatible with that of MPI. Other library solutions for load balancing are limited to certain types of parallel applications, as exemplified by Chombo [94] library for adaptive mesh refinement applications. The restriction on their applications makes AMPI's integrated, efficient and automated load balancing mechanism for a wide range of applications more desirable.

Chapter 8

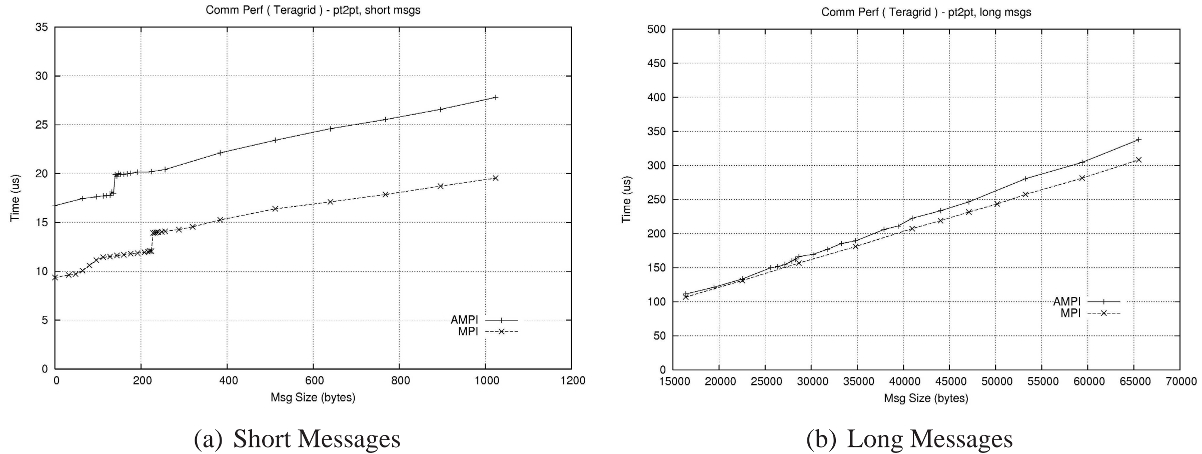
Evaluation of Adaptive MPI

The Message Passing Interface (MPI) has become the *de facto* standard for message passing programming. The MPI Standard specifies interface for a set of message passing functions. There are several very successful implementations from academia such as MPICH [95, 96], LAM/MPI [97, 98] and Open MPI [99], and almost all vendors of high performance computing platforms have their own native implementation of MPI.

8.1 AMPI Performance Evaluation

In this section, we present performance analysis of AMPI with various benchmarks and applications to demonstrate its advantages. Our main benchmarking platforms are the Turing Cluster with 640 dual Apple G5 nodes connected with Myrinet network and MPICH 1.2.6 at University of Illinois at Urbana-Champaign, NCSA's IA-64 TeraGrid Cluster with 888 dual Intel Itanium 2 nodes and Myrinet network installed with MPICH 1.2.5, NCSA's Tungsten Cluster with 1280 dual Intel Xeon nodes and Myrinet network with MPICH 1.2.5, and the Lemieux Cluster with 750 dual Alpha nodes and Quadrics network installed with MPICH 1.2.6 at Pittsburgh Supercomputer Center.

As a competitive implementation of MPI, AMPI offers performance benefits and functionality extensions to MPI applications, especially those with a dynamic nature. We now take a closer look at the comparison between AMPI and typical native MPI implementations, starting with a



(a) Short Messages (b) Long Messages

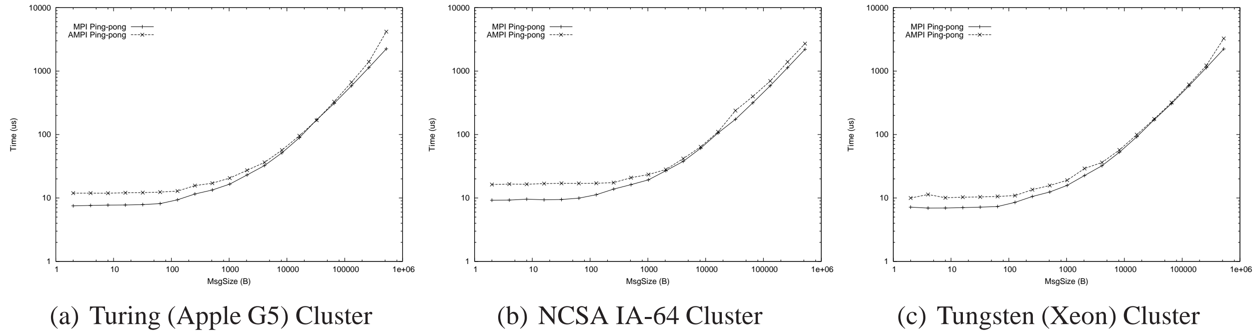
Figure 8.1: Point-to-point Performance on NCSA IA-64 Cluster

quantitative breakdown of the virtualization overheads of AMPI.

8.1.1 Virtualization Overheads

Because AMPI is implemented on top on the ARTS, which is typically implemented on top of native MPI (or the lowest level communication layer accessible to us), we do not expect to have better performance than native MPI on a ping-pong style microbenchmark. Point-to-point performance on the benchmarking platforms are listed in Figures 8.2(a), 8.2(b) and 8.2(c). To explain the breakdown of the virtualization overhead in the ping-pong benchmark, AMPI has a left-shift due to the 70+ byte AMPI message header, and a 2-4 microsecond increase in time for the short message latency due to thread context switch overhead as well as scheduling overhead (See Figure 8.1(a)). For longer messages, we pay the overhead of extra message copying in order to support migratable threads (See Figure 8.1(b)). Active research work is being carried out to reduce overhead for both situations.

In practice, the virtualization overheads are usually hidden via appropriate subtask assignment. From our experiences, the few microseconds of virtualization overhead is negligible when the average work driven by one message is at hundreds of microseconds level, which is achieved by the choice of number of VPs per processor. It should be noted that such “good choice” has a fairly



(a) Turing (Apple G5) Cluster

(b) NCSA IA-64 Cluster

(c) Tungsten (Xeon) Cluster

Figure 8.2: Point-to-point Communication Time

#PE	19	27	33	64	80	105	125	140	175	216	250	512
Native MPI	-	29.4	-	14.2	-	-	9.12	-	-	8.07	-	5.52
AMPI	42.4	30.5	24.7	15.6	12.6	10.9	10.8	10.6	9.39	8.63	7.55	5.46

Table 8.1: Timestep Time [ms] of 240^3 3D 7-point Stencil Calculation with AMPI vs. Native MPI on Lemieux

#PE	19	27	33	64	80	105	125	140	175	216	250	512
Native MPI	-	1786	-	702	-	-	367	-	-	212	-	91.7
AMPI	3633	1782	1367	697	711	705	364	387	385	207	250	92.1

Table 8.2: Timestep Time [ms] of 960^3 3D 7-point Stencil Calculation with AMPI v.s. Native MPI on NCSA IA-64 Cluster

large tolerance, since the virtualization overhead is not very sensitive to the number of VPs per processor. Moreover, the cost of supporting virtualization and coordinating the VPs is further offset by other benefits of virtualization. Therefore, it is safe to conclude that the functionality extensions and performance advantages of AMPI do not come at an undue price in basic performance.

8.1.2 Flexibility to Run

AMPI supports virtual MPI processes, thus giving the programmer the freedom to run multiple MPI processes on one physical processor. We illustrate this functionality extension with a more realistic benchmark, a 3D stencil-type calculation. The 3D stencil calculation is a multiple timestepping calculation involving a group of regions in a mesh. At each timestep, every region exchanges part of its data with its 6 immediate neighbors in 3D space and does some computation based on

the neighbors' data. This is a simplified model of many applications, like fluid dynamics or heat dispersion simulation, so it serves well the purpose of demonstration.

We show the flexibility of AMPI with a 240^3 3D 7-point stencil calculation. The algorithm in this particular benchmark divides a 240^3 block of data into k^3 partitions, each of which is a smaller cube assigned to an MPI processor. Natural expression of this algorithm requires k^3 number of processors to run on. This benchmark represents the type of applications that require specific number of processors.

On the Lemieux cluster, we first run the benchmark with native MPI. As described above, this program runs only on k^3 processors: 27, 64, 125, 216, 512, etc. Then, with virtualized AMPI, the program runs transparently on any given number of processors, exhibiting the flexibility that virtualization offers. The comparison between these two runs are listed in Tables 8.1 and 8.2. Note that on some arbitrary number of processors such as 19 and 80, the native MPI program cannot be launched, whereas AMPI runs the job with no difficulty. This flexibility has been proven to be very useful in real application development. For instance, during the development of the CSAR code, a specific software bug occurs only on 480 processors. Consequently, to debug it, the developers require 480 nodes on a parallel platform to launch their problematic run. With AMPI, the 480 processor run can be performed on a much smaller partition that is easier to get, offering the developer a great productivity advantage.

8.1.3 Adaptive Overlapping

The performance benefit of adaptive overlapping arises from the fact that the actual CPU overhead in a blocking communication operation is typically smaller than the total elapsed time. We show this with a multi-ping benchmark. In the benchmark, processor A sends multiple ping messages to processor B, which responds with a short pong message after it has received all of them. This communication pattern differs from the usual ping-pong benchmark in that it fills the pipeline on a message's path from sender to receiver: sender CPU, sender NIC, interconnect, receiver NIC, and receiver CPU. The performance from the multi-ping benchmark represents the limiting

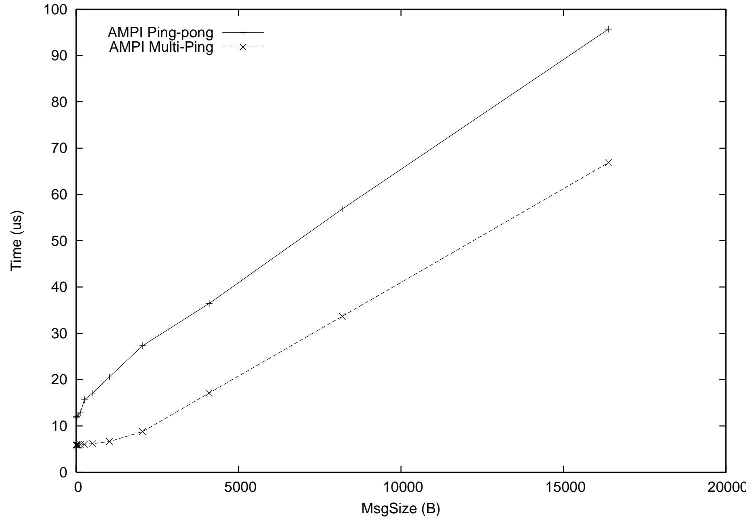


Figure 8.3: Performance of Ping-pong vs Multi-ping Benchmark on Turing (Apple G5) Cluster

K	VP=8	VP=16	VP=32	VP=64	MPICH
128	25.0	25.8	25.8	25.8	10.6
256	97.7	96.8	91.3	90.4	100.1
512	744.9	772.7	776.7	770.6	751.6
1024	7418	6545	5908	5894	7437

Table 8.3: Iteration Time [ms] of K^3 3D 7-point Stencil Calculation on 8 PEs of NCSA IA-64 Cluster

factor in the pipeline, namely the price for a point-to-point communication. The gap between its curve and the ping-pong benchmark's curve is the time spent waiting for the communication to complete. Figure 8.2(a) shows a large gap between the two curves, suggesting that much of the time usually attributed to communication can be utilized for useful computation. Many traditional MPI implementations, however, cannot take advantage of this gap easily.

In AMPI, several VPs can be mapped onto one physical processor. This design naturally allows adaptive overlapping of computation and communication without any additional programming complexity. When one VP is blocked at a communication call, it yields the CPU so that another VP residing on the same processor can take over and utilize it, as illustrated by the following stencil calculation benchmark.

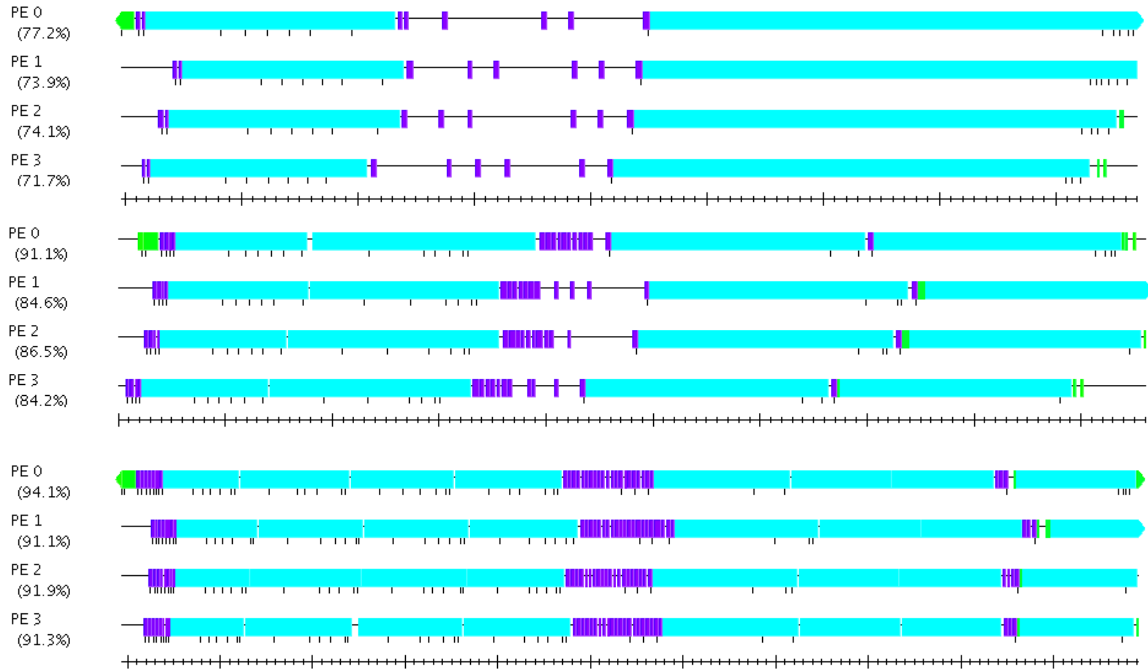


Figure 8.4: 7-point Stencil Timeline with 1, 2 and 4 VPs Per Processor

Table 8.3 shows the iteration time of 3D stencil calculations on 8 physical processors on an NCSA IA-64 Cluster. The calculations are of different sizes K^3, and with AMPI of various VP numbers. It can be observed that the overall performance increases when more VPs are mapped onto one processor in the given range. The underlying reason is illustrated by the projections visualization in Figure 8.4. The solid blocks represent computation and the gaps are idle time when CPU is waiting for communication to complete. As the number of VPs per processor increases, there are more opportunities for the smaller blocks (smaller pieces of computation on multiple VPs) to fill in the gaps, and consequently the CPU utilization increases.

It can also be observed that virtualization does not always result in performance improvement. As we introduce more VPs on one processor, virtualization overhead might overweigh the benefit from adaptive overlap and cause longer execution time, as we have discussed in previous sections.

Besides adaptive overlapping, the caching effect is also a favorable influence. VPs residing on the same processor can increase the spatial locality and turn some inter-processor communication

into intra-processor communication.

8.1.4 Automatic Load Balancing

Load balancing is one of the key factors for achieving high performance on large parallel machines when solving highly irregular problems. AMPI supports automatic measurement-based dynamic load balancing and thread migration based on the load balancing framework. In this section, we present the case studies of load balancing several MPI benchmarks and a real-world application.

NAS Parallel Benchmark (NPB) is a well known parallel benchmark suite. Its Multi-Zone version, LU-MZ, SP-MZ and BT-MZ, "solve discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions"[100]. The multi-zone version is characterized by partitioning of the problems on a coarse-grain level to expose more parallelism and to stress the need for balancing the computational load. Particularly on BT-MZ, the partitioning of the mesh is done such that the sizes of the zones span a significant range, creating imbalance in workload across processors. For such a problem, the load balancing requires two considerations, as suggested in [101]: careful zone grouping to minimize inter-processor communication and a multi-threading scheme to balance the computation workload across processors.

AMPI is naturally equipped with an automatic load balancing module to take these two aspects of a parallel program into consideration: communication load and computation load. The following results illustrate AMPI's effectiveness on load balancing BT-MZ.

In this benchmark, add a function call to trigger the automatic load balancing in AMPI run-time system. After 3 timesteps, when the run-time has collected sufficient information to advise the load balancer, the AMPI VPs are migrated from more heavily loaded processors onto more lightly loaded ones. The execution time is visualized in Figure 8.5.

When the number of processor increases for the same problem scale, we can make two observations. Firstly, the execution time without load balancing increases. BT-MZ creates workload imbalance by allocating different amounts of work among the processors, and with a larger number of processors, the degree of imbalance increases. Consequently, overall utilization drops.

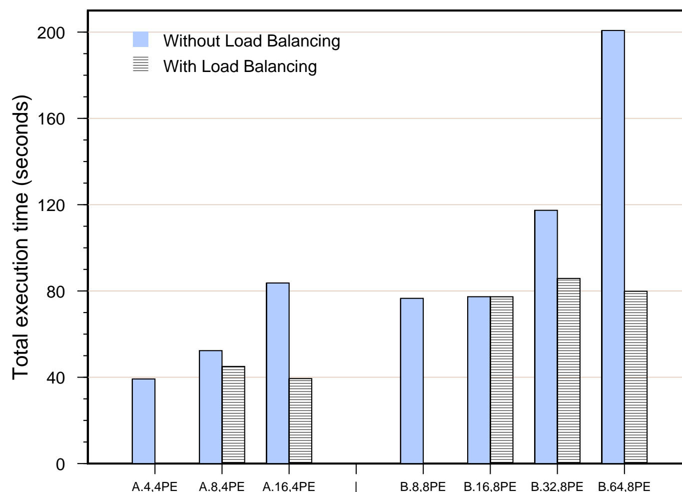


Figure 8.5: Load Balancing on NAS BT-MZ

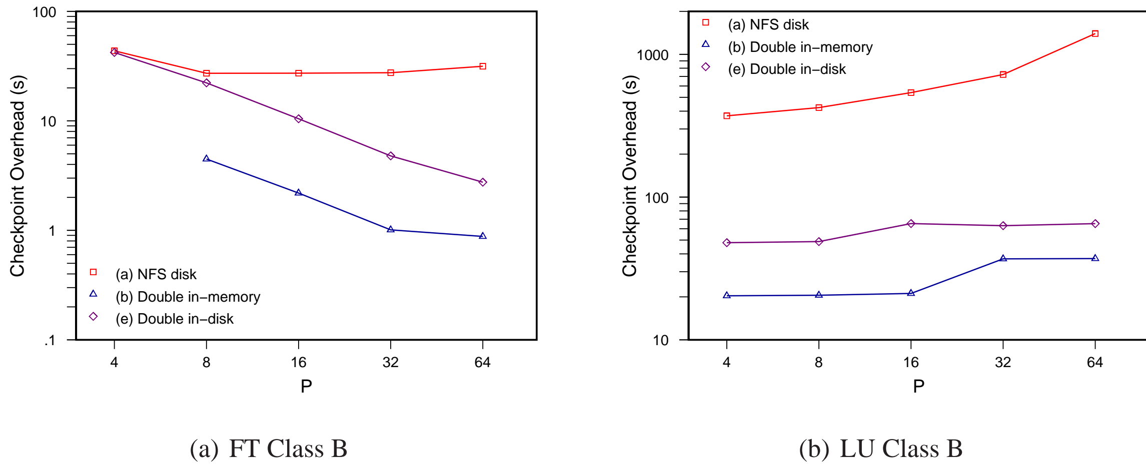
Secondly, with load balancing, having more VPs per processor allows the load balancing module to work more effectively, simply because there are more threads to move around if necessary. That is the reason that having number of VP much larger than number of P is recommended for the load balancer to be effective.

This benchmark demonstrates the effect of load balancing on applications with static load imbalance. This scenario is not uncommon. Handling uneven initial workload distribution and migrating the job away from faulty nodes are two cases that we have experienced where such load balancing is useful. For applications with a dynamically varying workload, the load balancer can be triggered periodically. The application example in Section 8.2.2 demonstrates this use.

8.1.5 Checkpoint Overhead

To illustrate the checkpoint overhead of AMPI, we perform our experiments with two NAS benchmarks FT and LU class B on the Turing cluster. The total amount of data to save is different: FT class B has nearly 2GB regardless of number of processor, while LU class B has saved data size roughly proportional to the number of processors, so the per processor data is nearly constant.

The results are shown in Figure 8.6(a) for FT and Figure 8.6(b) for LU. The x-axis is increasing



(a) FT Class B

(b) LU Class B

Figure 8.6: Checkpoint Overhead of NAS Benchmark on Turing Cluster

number of processors, the y-axis is checkpoint time in seconds, and both axes are logarithmic. In each figure, there are 3 curves representing 3 series of runs with the benchmark. The "NFS disk" scheme saves data to NFS disk. The "Double in-memory" and "Double in-disk" schemes save data in peers' memory or local disk to avoid the NFS disk bottleneck and use double checkpointing to ensure single fault tolerance.

For both benchmarks, the NFS disk scheme is the most expensive. Thanks to RAID disks, it scales to 4 and 8 processors for FT-B, but beyond that the NFS bottleneck becomes the limiting factor and the performance deteriorates as number of processors increases. The two double checkpointing runs utilize the fast Myrinet interconnect to transfer checkpoint data, and have similar scaling behavior. Because writing to local hard disk is not as fast as storing to peers' memory, we observe that the overhead of in-disk variation is always higher than its in-memory counterpart. With lower overhead from the in-memory scheme, we can checkpoint the program more often, and hence reduce the work lost since last checkpoint when a fault occurs.

In some cases external factors may constrain the available checkpointing options. In Figure 8.6(a) with FT class B, the double checkpoint scheme is unable to run on 4 processors, because the memory footprint for that specific benchmark (2GB) is too large for the machine. The same problem occurs on systems with relatively small per-node memory configuration, including

IBM's BlueGene/L. In this scenario, the user has two potential solutions. First, the user can use the in-disk variation of double checkpoint scheme. However, when a local disk is unavailable, as on BlueGene/L, the NFS disk checkpoint can still be used. Moreover, when the socket error detection required by the double checkpoint scheme is missing, the NFS disk scheme becomes the only choice.

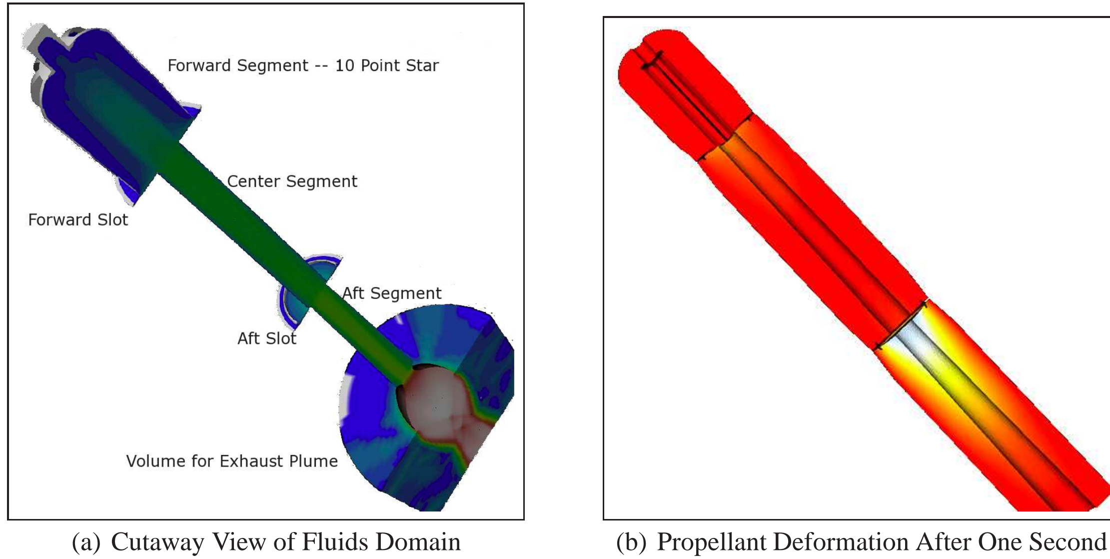
8.2 Application Case Study

In this section, we discuss two parallel applications which benefit from AMPI. These two projects are part of our collaboration with scientific and engineering researchers, and the benefits are not limited to performance gains.

8.2.1 Rocstar

The Center for Simulation of Advanced Rockets (CSAR) is an academic research organization funded by the Department of Energy and affiliated with the University of Illinois. The focus of CSAR is the accurate physical simulation of solid-propellant rockets, such as the Space Shuttle's solid rocket boosters. CSAR consists of several dozen faculty and professional staff from a number of different engineering and science departments. The main CSAR simulation code consists of several major components in various domains, including a fluid dynamics simulation, for the hot gas flowing through and out of the rocket; a surface burning model for the solid propellant; a nonmatching but fully-coupled fluid/solid interface; and finally a finite-element solid mechanics simulation for the solid propellant and rocket casing. Each one of these components - fluids, burning, interface, and solids - began as an independently developed parallel MPI program.

One of the most important early benefits CSAR found in using AMPI is the ability to run a partitioned set of input files on a different number of virtual processors than physical processors. For example, a CSAR developer was faced with an error in mesh motion that only appeared when



(a) Cutaway View of Fluids Domain

(b) Propellant Deformation After One Second

Figure 8.7: Titan IV Propellant Slumping Visualization

	AMPI						MPI
P	16	30	60	120	240	480	480
Time(s)	15.33	8.41	5.02	3.01	1.66	2.415	2.732

Table 8.4: *Rocstar* Performance Comparison of 480-processor Dataset for Titan IV SRMU Rocket Motor on Apple Cluster

a particular problem was partitioned for 480 processors. Finding and fixing the error was difficult, because a job for 480 physical processors can only be run after a long wait in the batch queue at a supercomputer center. Using AMPI, the developer was able to debug the problem interactively, using 480 virtual processors distributed over 32 physical processors of a local cluster, which made resolving the error much faster and easier.

Because each of the CSAR simulation components are developed independently, and each has its own parallel input format, there are difficult practical problems involved in simply preparing input meshes that are partitioned for the correct number of physical processors available. Using AMPI, CSAR developers can simply use a fixed number of virtual processors, which allows a wide range of physical processors to be used without repartitioning the problem's input files.

To demonstrate the performance benefits of virtualization using AMPI, we compared the performance of *Rocstar* using AMPI and MPICH/GM on different numbers of processors of the Turing Apple cluster with Myrinet interconnect at CSAR. Our test used a 480-processor dataset of the Titan IV SRMU Prequalification Motor #1. This motor exploded during a static test firing on April 1, 1991 due to excessive deformation of the aft propellant segment just below the aft joint slot [102]. Figure 8.7 shows a cutaway view of the fluids domain and the propellant deformation, obtained from *Rocstar*'s 3-D simulations at nearly one second after ignition for an incompressible neoHookean material model.

We ran *Rocstar* using AMPI (implemented on the native GM library) on various numbers of physical processors ranging from 16 to 480, and ran the same simulation with MPICH/GM on 480 processors. Table 8.4 shows the wall-clock times per iteration. The AMPI-based run outperformed the MPICH/GM-based by about 12% on 480 processors, demonstrating the efficiency of our AMPI implementation directly on top of the native GM library. Note that even better performance was obtained on 240 processors with two AMPI threads per physical processor. This virtualization allowed the AMPI run-time system to dynamically overlap communication with computation to exploit the otherwise idle CPU cycles while reducing interprocessor-communication overhead for the reduction in the number of physical processors, leading to a net performance gain for this test.

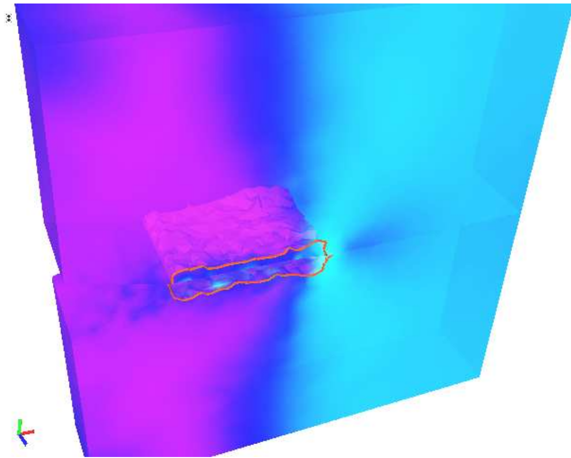


Figure 8.8: Fractography3D: Crack Propagation Visualization

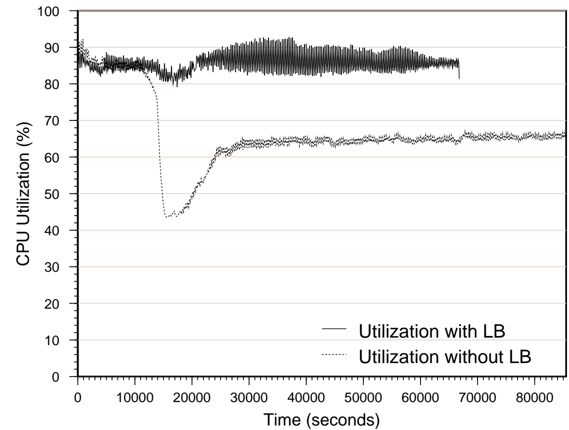


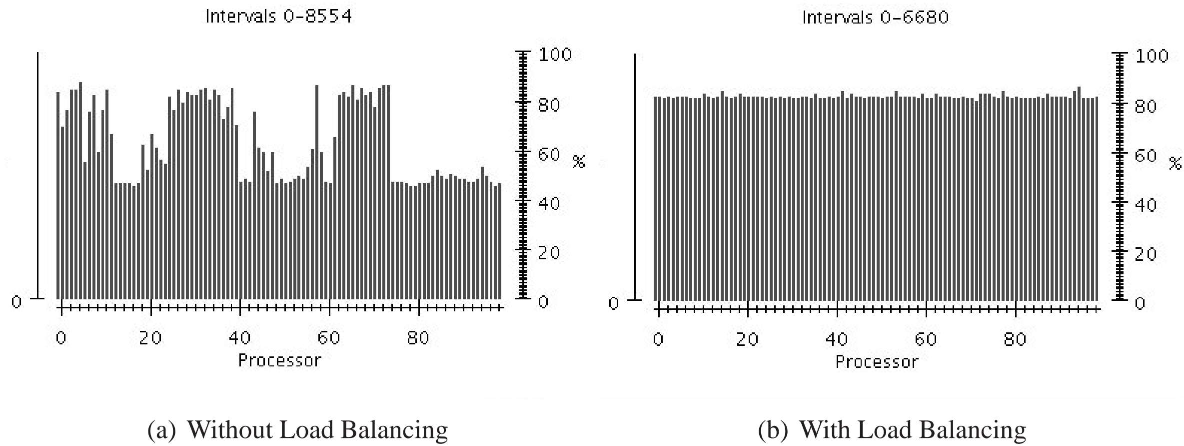
Figure 8.9: CPU Utilization Projections Graph of Fractography3D Over Time With and Without Load Balancing

8.2.2 Fractography3D

Fractography3d is a dynamic 3D crack propagation simulation program to simulate pressure-driven crack propagation in structures. It was developed by Professor Philippe Geubelle in the Department of Aerospace Engineering at the University of Illinois and his students in collaboration with our group. The Fractography3d code is implemented on the FEM framework [103] and AMPI.

For this experiment, the application simulates a force-induced crack propagating throughout a solid material and the conversion of the material from elastic to plastic in the zone along the crack, as illustrated in Figure 8.8. This crack propagation simulation was run with 1000 AMPI virtual processors on 100 processors of the Turing Cluster.

There are two factors that may contribute to the load imbalance in this simulation problem. When external force is applied to the material under study, the initial elastic state of the material converts to plastic along the wave propagation, which results in much heavier computation for the plastic elements. Second, to detect a crack in the domain, additional elements are inserted between some of the existing elements depending upon the forces exerted on the nodes. These added elements, which have zero volume, are called *cohesive elements*. At each iteration of the simulation, pressure exerted upon the plastic structure may propagate cracks, and therefore more



(a) Without Load Balancing (b) With Load Balancing

Figure 8.10: CPU Utilization Graphs of Fractography3D Across Processor With and Without Load Balancing

cohesive elements may have to be inserted. Thus, the amount of computation for some mesh partitions may increase during the simulation. This results in severe load imbalance.

The simulation without load balancing runs for 24 hours. The Projections view on CPU utilization over time is shown in the bottom curve of Figure 8.9. It can be seen that at around 1000 seconds, the application CPU utilization dropped from around 85% to only about 44%. This is due to the start of the conversion of elements from elastic to plastic along the crack, leading to load imbalance. As more elastic elements convert to plastic, the CPU utilization slowly increases until all elements have converted to plastic. The load imbalance can easily be seen in the CPU utilization vs. processor graph as shown in Figure 8.10(a). While some of the processors have an average CPU utilization as high as about 90%, some processors only have an average utilization of about 50% for the duration of the simulation.

The top curve of Figure 8.9 illustrates the results of automatic load balancing of the same crack propagation simulation in the view of overall CPU utilization over time. Load balancing is invoked every 500 time-steps of the simulation with a greedy strategy. The automatic load balancer uses run-time load and communication information obtained via a few instrumentation steps to migrate chunks from the overloaded processor to underloaded processors, leading to improved performance. As figure 8.9 shows, the overall CPU utilization on all processors throughout the

entire simulation stays around 80-90%. Figure 8.10(b) further illustrates that load balance has been improved over that shown in Figure 8.10(a). It can be seen that CPU utilization of at least 80% is achieved on all processors with little load variance. The simulation with load balancing now takes about 18.5 hours to complete, yielding approximately a 23% performance improvement.

Chapter 9

Conclusions

This thesis aims at improving parallel programming productivity by supporting multi-paradigm programming on top of an adaptive run-time system. In this final Chapter, we summarize the aspects of our research work completed in the thesis to achieve the goal, and conclude with future directions of further investigation.

As the first component in our framework, we explored a novel programming paradigm that allows clear expression of global view of control. This new paradigm combines the producer-consumer model and object-based parallel programming. It enables more efficient collaboration between parallel programmers and scientists/engineers who wish to parallelize applications in their specialty fields. It improves productivity with two primary features: higher-level abstraction and separation of parallelism specification and sequential component development.

To study the effectiveness of the proposed paradigm, we designed a reference language called Charisma. The philosophy behind Charisma is to allow explicit description of the program's global view of control by separating it from the sequential functions of the program. The new language features capability of producing efficient parallel code while reducing programming order for its target class of applications. It also provides facilities for developing reusable parallel libraries for both Charm++ and Charisma programming.

We developed a parallel programming environment for Charisma including a compiler that is capable of parsing the orchestration code and integrating sequential components to generate an optimized Charm++ program or library module. We improved the efficiency of the generated

program with various optimizations in the language implementation. We also discovered more cost-effective ways to coordinate message delivery and to eliminate data copying between local variables and generated messages. We conducted performance evaluation with various benchmark programs, and Charisma significantly cut SLOC while retaining performance.

To measure Charisma's productivity advantage over Charm++, we carried out a preliminary classroom study on productivity metrics beyond SLOC. The results show that to the target users, namely science and engineering application developers without significant parallel programming background, programming with Charisma is consistently easier than programming with Charm++ on two different applications. Feedback says that Charisma is "more intuitive" and takes less time to program with. It is also easier to add various features such as load balancing with Charisma than with Charm++.

Because parallel programming is more difficult than sequential programming, developing library modules and reusing code is an important way to improve productivity. A challenging design question is how to provide interfaces for library development with Charisma. Parallel libraries usually involve more complicated interactions between the calling main program and the library module than their sequential counterparts. We studied alternatives for library interface design, and developed capabilities for library module development for both Charisma and Charm++.

Beside the new paradigm represented by Charisma and the existing object-based message-driven paradigm represented by Charm++, we extend our support for existing prevalent paradigms. We designed generic mechanisms to support virtualized processors (VP) via migratable user-level threads embedded in communicating objects. This mechanism is portable under various environments. With this mechanism, we are able to develop adaptive implementations of both MPI and ARMCI, covering both the message passing and global address space paradigms.

Adaptive MPI combines the power of the ARTS and MPI programming model, enabling performance improvement for a wide range of applications and libraries. The work on Adaptive MPI was initiated with a partial implementation by previous group members, and I have continued exploring new research directions with our implementation. Significant research efforts have since

been put into more features and performance optimizations of AMPI. For instance, several alternative all-to-all schemes are integrated and applied according to the message size. Meanwhile, AMPI is ported to new HPC platforms to study its behavior on various architectures, such as IBM Blue Gene machines. With our work, AMPI has become a mature and competitive MPI implementation that delivers good performance for MPI programs with a dynamic nature. Adaptive ARMCI is a recent effort to integrate more paradigms, especially global address space languages, into the ARTS. We have implemented a functionally critical portion of the interface and conducted a preliminary performance study.

Charisma, AMPI and adaptive ARMCI are included as part of the latest release of Charm++ framework, together with a collection of stand-alone applications and library modules developed with Charisma and AMPI to demonstrate the effectiveness of our approach. User manuals are also prepared for Charisma and AMPI. We are confident that these tools will benefit future users with improved parallel programming productivity.

This thesis has an ambitious goal to inspire future research topics. With the design and implementation of our infrastructure for our high level language, the foundation has been laid for further investigations on improving parallel productivity. For example, current infrastructure for data dependence and control dependence analysis can be useful in important features such as out-of-core execution [104], critical path analysis [105], and other optimizations.

Looking forward, the future exploration based on the dataflow concept used by Charisma can include the following aspects. First, the same techniques that are used in orchestrate overall control flow in a parallel program can be applied to other fields. As an example, if a streaming-based dataflow applications such as the one described in [106] is to be developed with Charisma, streams of data can flow into nodes with inports and outports such that all modules can be pipelined and parallelization would be much easier. Secondly, during the static analysis, cost estimation at each node can be added to aid appropriate flow adjustment [107] to optimize overall performance. Thirdly, new generation performance analysis tools such as Projections [108] can take advantage of the information available in the orchestration code, since all the parallel flows are explicitly described.

Lastly, one can explore the future direction of evolving Charisma into a visual programming language [45] that looks more straightforward to programmers.

Future research directions also include supporting a wider array of global address space languages and their applications on the common run-time system. The ultimate goal for such implementations is porting various scientific and engineering applications and frameworks on top of the ARTS. We expect future research will demonstrate the advantage of the ARTS support by port existing benchmarks, libraries and applications, and show the performance gains from our implementations.

Appendix A

Charisma Manual

This manual describes Charisma, an orchestration language for migratable parallel objects.

A.1 Charisma Syntax

A Charisma program is composed of two parts: the orchestration code in a .or file, and sequential user code in C/C++ form.

A.1.1 Orchestration Code

The orchestration code in the .or file can be divided into two part. The header part contains information about the program, included external files, defines, and declaration of parallel constructs used in the code. The orchestration section is made up of statements that forms a global control flow of the parallel program. In the orchestration code, Charisma employs a macro dataflow approach; the statements produce and consume values, from which the control flows can be organized, and messages and method invocations generated.

Header Section

The very first line should give the name of the Charisma program with the `program` keyword.

```
    program jacobi
```

The `program` keyword can be replaced with `module`, which means that the output program is going to be a library module instead of a stand-alone program. Please refer to Section A.3 for more details.

Next, the programmer can include external code files in the generated code with keyword `include` with the filename without extension. For example, the following statement tells the Charisma compiler to look for header file "particles.h" to be included in the generated header file "jacobi.h" and to look for C/C++ code file "particles.[C/cc/cpp/cxx/c]" to be included in the generated C++ code file "jacobi.C".

```
include particles;
```

It is useful when there are source code that must precede the generated parallel code, such as basic data structure declaration.

After the `include` section is the `define` section, where environmental variables can be defined for Charisma. For example, to tell Charisma to generate additional code to enable the load balancing module, the programmer needs to define "ldb" in the orchestration code. Please refer to Section A.6 for details.

Declaration Section

Next comes the declaration section, where classes, objects and parameters are declared. A Charisma program is composed of multiple sets of parallel objects which are organized by the orchestration code. Different sets of objects can be instantiated from different class types. Therefore, we have to specify the class types and object instantiation. Also we need to specify the parameters (See Section A.1.1) to use in the orchestration statements.

A Charisma program or module has one "MainChare" class, and it does not require explicit instantiation since it is a singleton. The statement to declare MainChare looks like this:

```
class JacobiMain : MainChare;
```

For object arrays, we first need to declare the class types inherited from 1D object array, 2D object array, etc, and then instantiate from the class types. The dimensionality information of the object array is given in a pair of brackets with each dimension size separated by a comma.

```
class JacobiWorker : ChareArray1D;
obj workers : JacobiWorker[N];


class Cell : ChareArray3D;
obj cells : Cell[M,M,M];
```

Note that key word "class" is for class type derivation, and "obj" is for parallel object or object array instantiation. The above code segment declares a new class type JacobiWorker which is a 1D object array, (and the programmer is supposed to supply sequential code for it in files "JacobiWorker.h" and "JacobiWorker.C" (See Section A.1.2 for more details on sequential code). Object array "workers" is instantiated from "JacobiWorker" and has 16 elements.

The last part is orchestration parameter declaration. These parameters are used only in the orchestration code to connect input and output of orchestration statements, and their data type and size is declared here. More explanation of these parameters can be found in Section A.1.1.

```
param lb : double[N];
param rb : double[N];
```

With this, "lb" and "rb" are declared as parameters of that can be "connected" with local variables of double array with size of 512.

Orchestration Section

In the main body of orchestration code, the programmer describes the behavior and interaction of the elements of the object arrays using orchestration statements.

• **Foreach Statement**

The most common kind of parallelism is the invocation of a method across all elements in an object array. Charisma provides a *foreach* statement for specifying such parallelism. The keywords `foreach` and `end-foreach` forms an enclosure within which the parallel invocation is performed. The following code segment invokes the entry method `compute` on all the elements of array `myWorkers`.

```
foreach i in workers
  workers[i].compute();
end-foreach
```

• **Publish Statement and Produced/Consumed Parameters**

In the orchestration code, an object method invocation can have input and output (consumed and produced) parameters. Here is an orchestration statement that exemplifies the input and output of this object methods `workers.produceBorders` and `workers.compute`.

```
foreach i in workers
  (lb[i], rb[i]) <- workers[i].produceBorders();
  workers[i].compute(lb[i+1], rb[i-1]);

  (+error) <- workers[i].reduceData();
end-foreach
```

Here, the entry method `workers[i].produceBorders` produces (called *published* in Charisma) values of `lb[i], rb[i]`, enclosed in a pair of parentheses before the publishing sign "`<-`". In the second statement, function `workers[i].compute` consumes values of `lb[i+1], rb[i-1]`, just like normal function parameters. If a reduction operation is needed, the reduced parameter is marked with a "+" before it, like the `error` in the third statement.

A entry method can have arbitrary number of published (produced and reduced) values and consumed values. In addition to basic data types, each of these values can also be an object of arbitrary type. The values published by `A[i]` must have the index `i`, whereas values consumed can have the index `e(i)`, which is an index expression in the form of $\mathtt{i} \pm c$ where c is a constant. Although we have used different symbols (`p` and `q`) for the input and the output variables, they are allowed to overlap.

The parameters are produced and consumed in the program order. Namely, a parameter produced in an early statement will be consumed by the next consuming statement, but will no longer be visible to any consuming statement after that. Special rules involving loops are discussed later with loop statement.

• **Overlap Statement**

Complicated parallel programs usually have concurrent flows of control. To explicitly express this, Charisma provides a `overlap` keyword, whereby the programmer can fire multiple overlapping control flows. These flows may contain different number of steps or statements, and their execution should be independent of one another so that their progress can interleave with arbitrary order and always return correct results.

```
overlap
{
  foreach i in workers1
    (lb[i], rb[i]) <- workers1[i].produceBorders();
  end-foreach
  foreach i in workers1
    workers1[i].compute(lb[i+1], rb[i-1]);
  end-foreach
}
{
  foreach i in workers2
    (lb[i], rb[i]) <- workers2[i].compute(lb[i+1], rb[i-1]);
  end-foreach
}
end-overlap
```

This example shows an `overlap` statement where two blocks in curly brackets are executed in parallel. Their execution join back to one at the end mark of `end-overlap`.

- **Loop Statement**

Loops are supported with `for` statement and `while` statement. Here are two examples.

```
for iter = 0 to MAX_ITER
   workers.doWork();
end-for
```

```
while (err > epsilon)
   (+err) <- workers.doWork();
   MainChare.updateError(err);
end-while
```

The loop condition in `for` statement is independent from the main program; It simply tells the program to repeat the block for so many times. The loop condition in `while` statement is actually updated in the MainChare. In the above example, `err` and `epsilon` are both member variables of class `MainChare`,

and can be updated as the example shows. The programmer can active the "autoScalar" feature by including a "`define autoScalar;`" statement in the orchestration code. When autoScalar is enabled, Charisma will find all the scalars in the `.or` file, and create a local copy in the `MainChare`. Then every time the scalar is published by a statement, an update statement will automatically be inserted after that statement. The only thing that the programmer needs to do is to initialize the local scalar with a proper value.

Rules of connecting produced and consumed parameters concerning loops are natural. The first consuming statement will look for values produced by the last producing statement before the loop, for the first iteration. The last producing statement within the loop body, for the following iterations. At the last iteration, the last produced values will be disseminated to the code segment following the loop body. Within the loop body, program order holds.

```
for iter = 1 to MAX_ITER
  foreach i in workers
    (lb[i], rb[i]) <- workers[i].compute(lb[i+1], rb[i-1]);
  end-foreach
end-for
```

One special case is when one statement's produced parameter and consumed parameter overlaps. It must be noted that there is no dependency within the same `foreach` statement. In the above code segment, the values consumed `lb[i], rb[i]` by `worker[i]` will not come from its neighbors in this iteration. The rule is that the consumed values always originate from previous `foreach` statements or `foreach` statements from a previous loop iteration, and the published values are visible only to following `foreach` statements or `foreach` statements in following loop iterations.

- **Scatter and Gather Operation**

A collection of values produced by one object may be split and consumed by multiple object array elements for a scatter operation. Conversely, a collection of values from different objects can be gathered to be consumed by one object.

```
foreach i in A
  (points[i,*]) <- A[i].f(...);
end-foreach
foreach k,j in B
  (...) <- B[k,j].g(points[k,j]);
end-foreach
```

A wildcard dimension "*" in `A[i].f()`'s output `points` specifies that it will publish multiple data items. At the consuming side, each `B[k,j]` consumes only one point in the data, and therefore a scatter communication will be generated from `A` to `B`. For instance, `A[1]` will publish data `points[1,0..N-1]` to be consumed by multiple array objects `B[1,0..N-1]`.

```
foreach i,j in A
  (points[i,j]) <- A[i,j].f(...);
end-foreach
foreach k in B
  (...) <- B[k].g(points[*,k]);
end-foreach
```

Similar to the scatter example, if a wildcard dimension "*" is in the consumed parameter and the corresponding published parameter does not have a wildcard dimension, there is a gather operation generated from the publishing statement to the consuming statement. In the following code segment, each `A[i,j]` publishes a data point, then data points from `A[0..N-1,j]` are combined together to for the data to be consumed by `B[j]`.

Many communication patterns can be expressed with combination of orchestration statements. For more details, please refer to PPL technical report 06-18, "Charisma: Orchestrating Migratable Parallel Objects".

Last but not least, all the orchestration statements in the `.or` file together form the dependency graph. According to this dependency graph, the messages are created and the parallel program progresses. Therefore, the user is advised to put only parallel constructs that are driven by the data dependency into the orchestration code. Other elements such as local dependency should be coded in the sequential code.

A.1.2 Sequential Code

Sequential Files

The programmer supplies the sequential code for each class as necessary. The files should be named in the form of class name with appropriate file extension. The header file is not really an ANSI C header file. Instead, it is the sequential portion of the class's declaration. Charisma will generate the class declaration from the orchestration code, and incorporate the sequential portion in the final header file. For example, if a molecular dynamics simulation has the following classes (as declared in the orchestration code):

```
class MDMain : MainChare;

class Cell : ChareArray3D;

class CellPair : ChareArray6D;
```

The user is supposed to prepare the following sequential files for the classes: MDMain.h, MDMain.C, Cell.h, Cell.C, CellPair.h and CellPair.C, unless a class does not need sequential declaration and/or definition code. Please refer to the example in the Appendix.

For each class, a member function "void initialize(void)" can be defined and the generated constructor will automatically call it. This saves the trouble of explicitly call initialization code for each array object.

Producing and Consuming Functions

The C/C++ source code is nothing different than ordinary sequential source code, except for the producing/consuming part. For consumed parameters, a function treat them just like normal parameters passed in. To handle produced parameters, the sequential code needs to do two special things. First, the function should have extra parameter for output parameters. The parameter type is keyword `outport`, and the parameter name is the same as appeared in the orchestration code. Second, in the body of the function, the keyword `produce` is used to connect the orchestration parameter and the local variables whose value will be sent out, in a format of a function call, as follows.

```
produce(produced_parameter, local_variable[, size_of_array]);
```

When the parameter represents a data array, we need the additional `size_of_array` to specify the size of the data array.

The dimensionality of an orchestration parameter is divided into two parts: its dimension in the orches-

tration code, which is implied by the dimensionality of the object arrays the parameter is associated, and the local dimensionality, which is declared in the declaration section. The orchestration dimension is not explicitly declared anywhere, but it is derived from the object arrays. For instance, in the 1D Jacobi worker example, “lb” and “rb” has the same orchestration dimensionality of workers, namely 1D of size [16]. The local dimensionality is used when the parameter is associated with local variables in sequential code. Since “lb” and “rb” are declared to have the local type and dimension of “double [512]”, the producing statement should connect it with a local variable of “double [512]”.

```
void JacobiWorker::produceBorders(outport lb, outport rb){
  . . .
  produce(lb,localLB,512);
  produce(rb,localRB,512);
}
```

Special cases of the produced/consumed parameters involve scatter/gather operations. In scatter operation, since an additional dimension is implied in the produced parameter, we the `local_variable` should have additional dimension equal to the dimension over which the scatter is performed. Similarly, the input parameter in gather operation will have an additional dimension the same size of the dimension of the gather operation.

For reduction, one additional parameter of type char[] is added to specify the reduction operation. Built-in reduction operations are “+” (sum), “*” (product), “<” (minimum), “>” (maximum) for basic data types. For instance the following statements takes the sum of all local value of `result` and for output in `sum`.

```
reduce(sum, result, ``+'');
```

If the data type is a user-defined class, then you might use the function or operator defined to do the reduction. For example, assume we have a class called “Force”, and we have an “add” function (or a “+” operator) defined.

```
Force& Force::add(const Force& f);
```

In the reduction to sum all the local forces, we can use

```
reduce(sumForces, localForce, "add");
```

Miscellaneous Issues

In sequential code, the user can access the object's index by a keyword "thisIndex". The index of 1-D to 6-D object arrays are:

```
1D: thisIndex
2D: thisIndex.{x,y}
3D: thisIndex.{x,y,z}
4D: thisIndex.{w,x,y,z}
5D: thisIndex.{v,w,x,y,z}
6D: thisIndex.{x1,y1,z1,x2,y2,z2}
```

A.2 Building and Running a Charisma Program

There are two steps to build a Charisma program: generating Charm++ program from orchestration code, and building the Charm++ program.

1) Charisma compiler, currently named `orchc`, is used to compile the orchestration code (.or file) and integrate sequential code to generate a Charm++ program. The resultant Charm++ program usually consists of the following code files: Charm++ Interface file ([modulename].ci), header file ([modulename].h) and C++ source code file ([modulename].C). The command for this step is as follows.

```
> orchc [modulename].or
```

2) Charm++ compiler, charmc, is used to parse the Charm++ Interface (.ci) file, compile C/C++ code, and link and build the executable. The typical commands are:

```
> charmc [modulename].ci
> charmc [modulename].C -c
> charmc [modulename].o -o pgm -language charm++
```

Running the Charisma program is the same as running a Charm++ program, using Charm++'s job launcher `charmrun`. (On some platforms like CSE's Turing Cluster, use the customized job launcher `rjq` or `rj`.)

```
> charmrun pgm +p4
```

Please refer to Charm++'s manual and tutorial for more details of building and running a Charm++

program.

A.3 Support for Library Module

Charisma is capable of producing library code for reuse with another Charisma program. We explain this feature in the following section.

A.4 Writing Module Library

The programmer uses the keyword `module` instead of `program` in the header section of the orchestration code to tell the compiler that it is a library module. Following keyword `module` is the module name, then followed by a set of configuration variables in a pair parentheses. The configuration variables are used in creating instances of the library, for such info as problem size.

Following the first line, the library's input and output parameters are posted with keywords `inparam` and `outparam`.

```
module FFT3D(CHUNK, M, N);

inparam indata;

outparam outdata1,outdata2;
```

The body of the library is not very different from that of a normal program. It takes input parameters and produces out parameters, as posted in the header section.

A.5 Using Module Library

To use a Charisma module library, the programmer first needs to create an instance of the library. There are two steps: including the module and creating an instance.

```
use FFT3D;

library f1 : FFT3D(CHUNK=10,M=10,N=100);

library f2 : FFT3D(CHUNK=8,M=8,N=64);
```

The keyword `use` and the module name includes the module in the program, and the keyword `library` creates an instance with the instance name, followed by the module name with value assignment of config-

uration variables. These statements must appear in the declaration section before the library instance can be used in the main program's orchestration code.

Invoking the library is like calling a publish statement; the input and output parameters are the same, and the object name and function name are replaced with the library instance name and the keyword `call` connected with a colon.

```
(f1_outdata[*]) <- f1:call(f1_indata[*]);
```

Multiple instances can be created out of the same module. Their execution can interleave without interfering with one another.

A.6 Using Load Balancing Module

A.6.1 Coding

To activate load balancing module and prepare objects for migration, there are 3 things that needs to be added in Charisma code.

First, the programmer needs to inform Charisma about the load balancing with a "`define ldb;`" statement in the header section of the orchestration code. This will make Charisma generates extra Charm++ code to do load balancing such as `PUP` methods.

Second, the user has to provide a `PUP` function for each class with sequential data that needs to be moved when the object migrates. When choosing which data items to `pup`, the user has the flexibility to leave the dead data behind to save on communication overhead in migration. The syntax for the sequential `PUP` is similar to that in a Charm++ program. Please refer to the load balancing section in Charm++ manual for more information on `PUP` functions. A typical example would look like this in user's sequential `.C` file:

```
void JacobiWorker::sequentialPup(PUP::er& p){
  p|myLeft; p|myRight; p|myUpper; p|myLower;
  p|myIter;
  PUParray(p,(double *)localData,1000);
}
```

Thirdly, the user will make the call to invoke load balancing session in the orchestration code. The call

is `AtSync();` and it is invoked on all elements in an object array. The following example shows how to invoke load balancing session every 4th iteration in a for-loop.

```
for iter = 1 to 100
  // work work
  if(iter % 4 == 0) then
    foreach i in workers
      workers[i].AtSync();
    end-foreach
  end-if
end-for
```

If a while-loop is used instead of for-loop, then the test-condition in the `if` statement is a local variable in the program's MainChare. In the sequential code, the user can maintain a local variable called `iter` in MainChare and increment it every iteration.

A.6.2 Compiling and Running

Unless linked with load balancer modules, a Charisma program will not perform actual load balancing. The way to link in a load balancer module is adding `-module EveryLB` as a link-time option.

At run-time, the load balancer is specified in command line after the `+balancer` option. If the balancer name is incorrect, the job launcher will automatically print out all available load balancers. For instance, the following command uses `RotateLB`.

```
> ./charmrun ./pgm +p16 +balancer RotateLB
```

A.7 Handling Sparse Object Arrays

In Charisma, when we declare an object array, by default a dense array is created with all the elements populated. For instance, when we have the following declaration in the orchestration code, an array of NxNxN is created.

```
class Cell : ChareArray3D;
obj cells : Cell[N,N,N];
```

There are certain occasions when the programmer may need sparse object arrays, in which not all elements are created. An example is neighborhood force calculation in molecular dynamics application. We have a 3D array of Cell objects to hold the atom coordinates, and a 6D array of CellPair objects to perform pairwise force calculation between neighboring cells. In this case, not all elements in the 6D array of CellPair are necessary in the program. Only those which represent two immediately neighboring cells are needed for the force calculation. In this case, Charisma provides flexibility of declaring a sparse object array, with a `sparse` keyword following the object array declaration, as follows.

```
class CellPair : ChareArray6D;
obj cellpairs : CellPair[N,N,N,N,N,N],sparse;
```

Then the programmer is expected to supply a sequential function with the name `getIndex_ARRAYNAME` to generate a list of selected indices of the elements to create. As an example, the following function essentially tells the system to generate all the NxNxNxNxNxN elements for the 6D array.

```
void getIndex_cellpairs(CkVec<CkArrayIndex6D>& vec){
  int i,j,k,l,m,n;
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      for(k=0;k<N;k++)
        for(l=0;l<N;l++)
          for(m=0;m<N;m++)
            for(n=0;n<N;n++)
              vec.push_back(CkArrayIndex6D(i,j,k,l,m,n));
}
```

Appendix B

LeanCP Orchestration Code

In this Appendix we show the orchestration code for Charisma version of LeanCP. The code is accompanied with 19 sequential C++ code files, which are not shown here. The orchestration code is composed of three sections, which are listed here with brief explanations. Some code segments are omitted to keep the layout clear.

B.1 Header Section

In the first section, the name of the Charisma program is given after the `program` keyword. For library modules, the keyword `module` is used instead.

Immediately following that, the programmer can include external files in the generated code with the keyword `include` and the filename to be included. Unlike in C/C++, `include` goes beyond including header files. It can also be used to include Charm++'s interface definition file in `.ci` format. For example, statement "`include FFTGroup`" integrates `FFTGroup.h`, `FFTGroup.ci` and `FFTGroup.C` in the generated files for class `FFTGroup`, which defines a per-processor object array that optimizes memory performance for FFT operations.

```
program leanCP

include params;
include FFTGroup;
```

B.2 Declaration Section

Next in the orchestration code is declaration section, where the classes, objects and object arrays, and parameter variables are declared. For each class, the programmer uses the `class` keyword to specify its inheritance from the base class such as generic 1D object array. Similarly, the keyword `obj` is used to instantiate objects and object arrays from the class type and dimensionality information. The third components in this section declares parameter variables that will serve as produced/consumed parameters in the orchestration statements.

```
class leanCPMain : MainChare;
class State_GSpacePlane : ChareArray2D;
class State_RealSpacePlane : ChareArray2D;
class Rho_RealSpacePlane : ChareArray1D;
class Rho_GSpacePlane : ChareArray1D;
. . .

obj main : leanCPMain;
obj gSpacePlane : State_GSpacePlane [nStates, sizeY];
obj realSpacePlane : State_RealSpacePlane [nStates, sizeZ];
obj rhoReal : Rho_RealSpacePlane [sizeZ];
obj rhoG : Rho_GSpacePlane [sizeY];
. . .

param cSGToSReal : complex[sizeX];
param cSRealToSG : complex[sizeX];
param rSRealToRReal : double[sizeX*sizeY];
param rRRealToSReal : double[sizeX*sizeY];
param iRRealToRRHart : int;
param cRRealToRG : complex[sizeX];
param cRGToRReal : complex[sizeX];
param cRGToRRealdiv : complex[3*sizeX];
. . .
```

B.3 Orchestration Section

Orchestration section is the core of an orchestration program, as it comprises all the orchestration statements that describe the behavior of and interactions among the parallel objects in the program. This section begins with a `begin` keyword and ends with an `end` keyword.

```
begin
  // initialization
  leanCPMain.init();
  for iter = 1 to MAXITER
    foreach i,y in gSpacePlane
      (cSGToSReal[i,y,*],cSGToGPP[i,y])
          <- gSpacePlane[i,y].PhaseI();
    end-foreach
    overlap
    { // density calculation
      foreach i,z in realSpacePlane
        (+rSRealToRReal[z])
            <- realSpacePlane[i,z].PhaseII(cSGToSReal[i,*,z]);
      end-foreach
      foreach z in rhoReal
        (cRRealToRG[*,z],+iRRealToRRHart)
            <- rhoReal[z].PhaseIII1(rSRealToRReal[z]);
      end-foreach
      foreach y in rhoG
        (cRGToRRealdiv[y,*],cRGToRGHart[y])
            <- rhoG[y].PhaseIV1(cRRealToRG[y,*]);
      end-foreach
      . . .
    }
    { // Non-Local energy
      foreach i,y in particlePlane
        (cGPPToRPP[i,y,*])
            <- particlePlane[i,y].PhaseIX1(cSGToGPP[i,y]);
      end-foreach
      foreach i,z in realParticlePlane
        (cRPPToGPP[i,*,z])
            <- realParticlePlane[i,z].PhaseIX(cGPPToRPP[i,*,z]);
      end-foreach
      foreach i,y in particlePlane
        (cGPPToSG[i,y])
            <- particlePlane[i,y].PhaseIX2(cRPPToGPP[i,y,*]);
      end-foreach
    }
    end-overlap

    // integration
    foreach i,y in gSpacePlane
      gSpacePlane[i,y].PhaseVI(cSRealToSG[i,y,*],cGPPToSG[i,y]);
    end-foreach
    . . .
  end-for
  leanCPMain.final();
end
```

Appendix C

AMPI Extension API

Beyond what is specified in the MPI Standard, AMPI defines a set of extensions, including an extra option for running with virtual processes, and several additional calls for various extension functionalities. The extension functions' names start with *AMPI_* instead of the usual *MPI_*, and their syntax and behavior are explained as follows.

C.1 Running with Virtual Processes

When the user run an AMPI program the usual way with the *-np P* option, the program is launched on *P* processors, with one virtual process on each processor, hence without any adaptivity support. AMPI provides a *+vp VP* option to specify to total number of virtual processes to run on the *P* processors.

- *ampirun -np P pgm*: launch program *pgm* on *P* processors, with one virtual process on each processor. (*P* = *VP*)
- *ampirun -np P pgm +vp VP*: launch program *pgm* on *P* processors, with *VP* virtual processes.

C.2 Automatic Load Balancing Interface

AMPI provides three load balancing function calls as extension of the MPI Standard.

- *void AMPI_Migrate(void)*: collective call that signals possible load balancing point. This call suspends the execution of the virtual processes, even though the actual migration may or may not happen, depending on the LB Manager's decision.

- *void AMPI_Async_Migrate(void)*: collective call that starts load balancing session while allowing the application to continue, such that load balancing overlaps with computation. When the load balancing decision is available, the threads could be migrated asynchronously.

- *void AMPI_Setmigratable(int comm, int mig)*: collective call that enables or disables load balancing in a communicator according to the value of *mig*.

- *void AMPI_Migrateto(int destPE)*: local call to force migration of the calling VP to *destPE* without being directed by the LB Manager.

C.3 Automatic Checkpointing Interface

AMPI extension for checkpointing on disk and in peers' system memory includes two function calls.

- *void AMPI_Checkpoint(char *dname)*: collective call that initiates on-disk checkpointing of the current program into directory given by *dname*.

- *void AMPI_MemCheckpoint()*: collective call that initiates in-memory checkpointing. The peers on which the thread's data is saved are chosen by the run-time system.

C.4 Asynchronous Collective Communication Interface

Collective calls involves many or all MPI processes and are time-consuming. AMPI offers the flexibility of making non-blocking collective communication calls such as reduction and all-to-all. These calls help the user exploit the large gap between the elapsed time and CPU time of collective operations. This features is especially helpful in the context of our adaptive implementation. When virtual processes in a communicator are blocked on a collective operation, the CPUs they reside on can be used by other virtual processes on the same physical processors.

The function interface is very much like the blocking counterpart, with an extra *MPI_Request* request* parameter returning the request handler.

- *void AMPI_Ireduce(..., MPI_Request* request)*

- *void AMPI_Iallreduce(..., MPI_Request* request)*
- *void AMPI_Ialltoall(..., MPI_Request* request)*
- *void AMPI_Iallgather(..., MPI_Request* request)*

References

[1] M. T. Heath and W. A. Dick. Virtual rocketry: Rocket science meets computer science. *IEEE Comptational Science and Engineering*, 5(1):16–26, 1998.

[2] I. D. Parsons, P. V. S. Alavilli, A. Namazifard, J. Hales, A. Acharya, F. Najjar, D. Tafti, and X. Jiao. Loosely coupled simulation of solid rocket moters. In *Fifth National Congress on Computational Mechanics*, Boulder, Colorado, August 1999.

[3] An Overview of the Blue Gene/L Supercomputer. In *Supercomputing 2002 Technical Papers*, Baltimore, Maryland, 2002. The Blue Gene/L Team, IBM and Lawrence Livermore National Laboratory.

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

[5] DARPA. High Productivity Computing System Program Website. http://www.highproductivity.org/.

[6] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[7] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[8] Orion Sky Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[9] Chao Huang. System support for checkpoint and restart of charm++ and ampi applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[10] Sayantan Chakravorty and L. V. Kale. A fault tolerance protocol with fast fault recovery. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[11] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.

[12] Sameer Kumar, Chao Huang, Gengbin Zheng, Eric Bohm, Abhinav Bhatele, James C. Phillips, Hao Yu, and Laxmikant V. Kalé. Scalable Molecular Dynamics with NAMD on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems (to appear)*, 52(1/2), 2007.

[13] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[14] Eric Bohm, Glenn J. Martyna, Abhinav Bhatele, Sameer Kumar, Laxmikant V. Kale, John A. Gunnels, and Mark E. Tuckerman. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems (to appear)*, 52(1/2), 2007.

[15] Ramkumar V. Vadali, Yan Shi, Sameer Kumar, L. V. Kale, Mark E. Tuckerman, and Glenn J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16):2006–2022, Oct. 2004.

[16] Xiangmin Jiao, Gengbin Zheng, Orion Lawlor, Phil Alexander, Mike Campbell, Michael Heath, and Robert Fiedler. An integration framework for simulations of solid rocket motors. In *41st AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, Tucson, Arizona, July 2005.

[17] Filippo Gioachin, Amit Sharma, Sayantan Chakravorty, Celso Mendes, Laxmikant V. Kale, and

Thomas R. Quinn. Scalable cosmology simulations on parallel machines. In *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[18] Gengbin Zheng, Michael S. Breitenfeld, Hari Govind, Philippe Geubelle, and Laxmikant V. Kale. Automatic dynamic load balancing for a crack propagation application. Technical Report 06-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, June 2006.

[19] Kai Wang, Anthony Chang, Jonathan A. Dantzig, and Laxmikant V. Kale. Parallelization of level set methods for solving solidification problems. Technical Report 05-22, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 2005.

[20] Attila Gursoy and L.V. Kalé. Performance and modularity benefits of messagedriven execution. *Journal of Parallel and Distributed Computing*, 64:461–480, 2004.

[21] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *J. Supercomputing*, (10):197–220, 1996.

[22] Tarek El-Ghazawi and Francois Cantonnet. Upc performance and potential: a npb experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[23] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques (PACT 2004)*, Antibes Juan-les-Pins, France, October 2004.

[24] Jayant DeSouza and Laxmikant 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.

[25] L. V. Kale and Attila Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 738–743, San Francisco, California, USA, February 1995.

[26] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.

[27] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[28] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[29] T.A. Darden, D.M. York, and L.G. Pedersen. Particle mesh Ewald. An N·log(N) method for Ewald sums in large systems. *JCP*, 98:10089–10092, 1993.

[30] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[31] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[32] B. Massingill, T. Mattson, and B. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP99)*, 1999.

[33] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, Boston, 2004.

[34] Gul A. Agha and Wooyoung Kim. Actors: a unifying model for parallel and distributed computing. *J. Syst. Archit.*, 45(15):1263–1277, 1999.

[35] Gul Agha and Carl Hewitt. *Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism*, volume 206 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, Berlin-Heidelberg-New York, October 1985.

[36] L. V. Kalé, Mark Hills, and Chao Huang. An orchestration language for parallel objects. In *Pro-*

ceedings of Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 04), Houston, Texas, October 2004.

[37] Jean-Luc Gaudiot and Liang-Teh Lee. Multiprocessor systems programming in a high-level data-flow language. In *Proceedings of the Parallel Architectures and Languages Europe, Volume I: Parallel Architectures PARLE*, pages 134–151, London, UK, 1987. Springer-Verlag.

[38] G. R. Gao. An efficient hybrid dataflow architecture model. *J. Parallel Distrib. Comput.*, 19(4):293–307, 1993.

[39] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[40] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[41] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *FPCA'91 — Conference on Functional Programming Languages and Computer Architectures*, volume 523, pages 538–568, Harvard, MA, 1991. Springer-Verlag.

[42] I. Foster and K.M. Chandy. FORTRAN M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 25(1), 1995.

[43] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[44] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[45] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

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

[47] Nasim Mahmood, Guosheng Deng, and James C. Browne. Compositional development of parallel programs. In *Lecture Notes in Computer Sciences*, volume 2958, pages 109–126, College Station, Texas, USA, October 2003. Springer-Verlag.

[48] Young Yoon, James C. Browne, Mathew Crocker, Samit Jain, and Nasim Mahmood. Productivity and performance through components: the asci sweep3d application: Research articles. *Concurrency and Computation: Practice and Experience*, 19(5):721–742, 2007.

[49] Adam Beguelin, Jack J. Dongarra, George Al Geist, Robert Manchek, and Keith Moore. HeNCE: a heterogeneous network computing environment. *Scientific Programming*, 3(1):49–60, Spring 1994.

[50] Peter Newton and James C. Browne. The code 2.0 graphical parallel programming language. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 167–177, New York, NY, USA, 1992. ACM Press.

[51] James C. Browne, Jack Dongarra, Syed I. Hyder, Keith Moore, and Peter Newton. Experiences with CODE and HeNCE in visual programming for parallel computing. *IEEE Parallel and Distributed Technology*, 1995.

[52] P. Newton and J. Dongarra. Overview of VPE: A visual environment for message-passing. In *Proceedings of the 4th Heterogeneous Computing Workshop*, 1995.

[53] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[54] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[55] 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.

[56] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the nas mg benchmark across parallel languages and architectures. In *Supercomputing '00: Proceedings of the*

2000 ACM/IEEE conference on Supercomputing (CDROM), page 46, Washington, DC, USA, 2000. IEEE Computer Society.

[57] 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.

[58] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Marvin Zelkowitz, Victor Basili, Sima Asgari, Lorin Hochstein, Jeff Hollingsworth, and Taiga Nakamura. Measuring productivity on high performance computers. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, page 6, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Robert W. Numrich, Lorin Hochstein, and Victor R. Basili. A metric space for productivity measurement in software development. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 13–16, New York, NY, USA, 2005. ACM Press.

[61] Andrew Funk, Victor Basili, Lorin Hochstein, and Jeremy Kepner. Application of a development time productivity metric to parallel software development. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 8–12, New York, NY, USA, 2005. ACM Press.

[62] K.P. Eurenius, D.C. Chatfield, B.R. Brooks, and M. Hodoscek. Enzyme mechanism with hybrid quantum and molecular mechanical potentials i. theory. *International Journal of Quantum Chemistry*, 60:1189, (1996).

[63] H. Goldstein. *Classical Mechanics*. Addison-Wesley, Reading, MA, (1980).

[64] M. E. Tuckerman, P. J. Ungar, T. von Rosenvinge, and M. L. Klein. Ab initio molecular dynamics simulations. *Journal of Physical Chemistry*, 100:12878, (1996).

[65] R. Car and M. Parrinello. Unified approach for molecular dynamics and density-functional theory. *Physical Review Letters*, 55:2471–2474, (1985).

[66] R. G. Parr and W. Yang. *Density Functional Theory of atoms and molecules*. Oxford University Press, Oxford, (1989).

[67] M.P. Bendsoe and O. Sigmund. *Topology Optimization*. Springer, February 2004.

[68] T. Borrvall and J. Petersson. Large-scale topology optimization in 3d using parallel computing. *Computer Methods in Applied Mechanics and Engineering*, 190(46):6201–6229(29), September 2001.

[69] G. DeRose Jr. and A. Diaz. Hierarchical solution of large-scale three-dimensional topology optimization problems. In *Proceedings Of The 1996 ASME Design Engineering Technical Conference and Computer in Engineering Conference*, Irvine, CA, August 1996.

[70] Jarek Nieplocha and Bryan Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *J. Rolim eat al. (eds.) Parallel and Distributed Processing, Springer Verlag LNCS 1586*, 1999.

[71] Gengbin Zheng, Orion Sky Lawlor, and Laxmikant V. Kalé. Multiple flows of control in migratable parallel programs. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, pages 435–444, Columbus, Ohio, August 2006. IEEE Computer Society.

[72] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[73] Neelam Saboo, Arun Kumar Singla, Joshua Mostkoff Unger, and L. V. Kalé. Emulating petaflops machines and blue gene. In *Workshop on Massively Parallel Processing (IPDPS'01)*, San Francisco, CA, April 2001.

[74] Joy Mukherjee and Srinidhi Varadarajan. Weaves: A framework for reconfigurable programming. *International Journal of Parallel Programming*, 33(2-3):279–305, 2005.

[75] John R. Levine. *Linkers and Loaders*. Morgan-Kauffman, 1999.

[76] Karthikeyan Mahesh. Ampizer: An mpi-ampi translator. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champiagn, 2001.

[77] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science, pages 141–154, Ithaca, NY, USA, August 1994. Springer-Verlag.

[78] Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *Lecture Notes in Computer Science: Proc. of Joint Modular Languages Conference (JMLC03)*, volume 2789, pages 214–223. Springer-Verlag, June 2003.

[79] Daniel Quinlan, Qing Yi, Gary Kumfert, Thomas Epperly, and Tamara Dahlgren. Toward the automated generation of components from existing source code.

[80] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM^2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

[81] Milind A. Bhandarkar. *Charisma: A Component Architecture for Parallel Programming*. PhD thesis, Dept. of Computer Science, University of Illinois, 2002.

[82] Nilesh Choudhury, Yogesh Mehta, Terry L. Wilmarth, Eric J. Bohm, and Laxmikant V. Kalé. Scaling an optimistic parallel simulation of large-scale interconnection networks. In *Proceedings of the Winter Simulation Conference*, 2005.

[83] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, volume 33, pages 183–207, 2005.

[84] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin

Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235.

[85] Hong Tang, Kai Shen, and Tao Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems*, 22(4):673–700, 2000.

[86] Gregory A. Koenig and Laxmikant V. Kale. Optimizing distributed application performance using dynamic grid topology-aware load balancing. In *21st IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[87] Gregory A. Koenig and Laxmikant V. Kale. Using message-driven objects to mask latency in grid computing applications. In *19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[88] Rohit Fernandes, Keshav Pingali, and Paul Stodghill. Mobile mpi programs in computational grids. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 22–31, New York, NY, USA, 2006. ACM Press.

[89] N. Drosinos and N. Koziris. Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium 2004 (CDROM)*, page 10, Santa Fe, New Mexico, 2004.

[90] Lorna Smith and Mark Bull. Development of mixed mode mpi / openmp applications. *Scientific Programming*, 9(2-3/2001):83–98. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.

[91] R. Wismller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Rder, and G. Stellner. The tool-set project: Towards an integrated tool environment for parallel programming. In *Proceedings of Second Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97, Koblenz, Germany*, pages 9–16. Verlag Dietmar Folbach, September 1997.

[92] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[93] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challenges in dynamic load balancing. *Appl. Numer. Math.*, 52(2–3):133–152, 2005.

[94] P. Colella, D.T. Graves, T.J. Ligocki, D.F. Martin, D. Modiano, D.B. Serafini, and B. Van Straalen. Chombo Software Package for AMR Applications Design Document, 2003. http://seesar.lbl.gov/anag/chombo/ChomboDesign-1.4.pdf.

[95] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. MPICH: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[96] W. Gropp and E. Lusk. The MPI communication library: its design and a portable implementation. In *Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi*, pages 160–165, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[97] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[98] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[99] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[100] Rob F. Van der Wijngaart and Haoqiang Jin. Nas parallel benchmarks, multi-zone versions. Technical Report NAS Technical Report NAS-03-010, July 2003.

[101] Haoqiang Jin and Rob F. Van der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[102] W. G. Wilson, J. M. Anderson, and M. Vander Meyden. Titan IV SRMU PQM-1 overview, 1992. AIAA Paper 92-3819.

[103] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[104] Mani Potnuru. Automatic out-of-core exceution support for charm++. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[105] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 366–373, 1988.

[106] S. Chandrasekaran and M. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proceedings of VLDB*, 2004.

[107] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 202–211, New York, NY, USA, 1986. ACM Press.

[108] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.

Author's Biography

Chao Huang was born in Xianning, Hubei, China, and grew up in that small town until he went to high school as one of the 26 outstanding students selected nationwide to form the National Science Class in Shanghai.

In 1997, he entered Tsinghua University in Beijing, China, and was awarded merit-based scholarship for four consecutive years. In 2001, he received a B.E. degree in Computer Science from Tsinghua University.

He then started his graduate study at the University of Illinois at Urbana-Champaign. He joined the Parallel Programming Laboratory as a research assistant under the guidance of Professor L. V. Kalé. Utilizing Charm++'s run-time system, he worked to improve and optimize an adaptive implementation of the MPI standard called *AMPI*. In 2004, he earned an M.S. degree in Computer Science. His master thesis is on System Support for Checkpoint/Restart of Charm++ and AMPI Applications. Subsequently, he started his investigation on the topic of high-level language that allows expression of overall flow of control in complicated parallel programs, which became his Ph.D. thesis topic.