X10 for High-Performance Scientific Computing

Josh Milthorpe

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

June 2015

© Josh Milthorpe 2015

Except where otherwise indicated, this thesis is my own original work.

Josh Milthorpe
8 March 2015

To Amy, Bridie and Felix
and the good times ahead.

Acknowledgments

> If you were successful, somebody along the line gave you some help. There was a great teacher somewhere in your life.
>
> – Barack Obama, 13 July 2012, Roanoke, Virginia.

There have been many great teachers in my life. Foremost among these in recent years have been my supervisor, Alistair Rendell; Steve Blackburn; and David Grove. These three have favoured me with countless hours of discussion, review and advice, for which I will always be grateful.

This research would not have been possible without the support of many researchers within IBM. Thanks go specifically to Olivier Tardieu, Vijay Saraswat, David Cunningham, Igor Peshansky, Ben Herta, Mandana Vaziri and Mikio Takeuchi.

For valuable discussions, ideas and difficult questions, I am indebted to: Bradford Chamberlain, John Mellor-Crummey, Vivek Sarkar, Jisheng Zhao, Andrey Blizniuk, Daniel Frampton, Peter Gill, Michelle Mills-Strout, Thomas Huber, Vivek Kumar, Taweetham Limpanuparb, Peter Strazdins and Uwe Zimmer.

For moral support and broader intellectual contributions, I would like to thank Evan Hynd, Ben Swift, Torben Sko, Ian Wood, Ting Cao, Brian Lee, Xi Yang, Nimalan Nandapalan, Joseph Antony, Pete Peerapong Janes, Jie Cai, James Barker, Lawrence Murray, John Taylor, Shin-Ho Chung, Henry Gardner, Chris Johnson, Lynette Johns-Boast, Richard Jones, Andrew Holmes and my sister Naomi Milthorpe.

The ANUChem application codes benefited from significant contributions by V. Ganesh, Andrew Haigh and Taweetham Limpanuparb – thank you all for the code, and the fun we had writing it. Andrew Gilbert and Taweetham Limpanuparb provided the comparison timings for Q-Chem used in chapter 4.

I have received generous financial support in the form of an Australian Postgraduate Award from the Australian Government and a supplementary scholarship from the ANU College of Engineering and Computer Science. Computing facilities to support this work were provided by IBM and the ANU under Australian Research Council Linkage Grant LP0989872, and by the NCI National Facility at the ANU.

Finally I would like to thank all my family and friends who have tirelessly supported me through long years. In particular I thank my parents Robyn and John, who first set me on the path that led me to a research career and have supported me again through this second childhood; and most of all Amy, for sticking with me and reminding me that there's always more to life.

Abstract

High performance computing is a key technology that enables large-scale physical simulation in modern science. While great advances have been made in methods and algorithms for scientific computing, the most commonly used programming models encourage a fragmented view of computation that maps poorly to the underlying computer architecture.

Scientific applications typically manifest *physical locality*, which means that interactions between entities or events that are nearby in space or time are stronger than more distant interactions. *Linear-scaling methods* exploit physical locality by approximating distant interactions, to reduce computational complexity so that cost is proportional to system size. In these methods, the computation required for each portion of the system is different depending on that portion's contribution to the overall result. To support productive development, application programmers need programming models that cleanly map aspects of the physical system being simulated to the underlying computer architecture while also supporting the irregular workloads that arise from the fragmentation of a physical system.

X10 is a new programming language for high-performance computing that uses the asynchronous partitioned global address space (APGAS) model, which combines explicit representation of locality with asynchronous task parallelism. This thesis argues that the X10 language is well suited to expressing the algorithmic properties of locality and irregular parallelism that are common to many methods for physical simulation.

The work reported in this thesis was part of a co-design effort involving researchers at IBM and ANU in which two significant computational chemistry codes were developed in X10, with an aim to improve the expressiveness and performance of the language. The first is a Hartree–Fock electronic structure code, implemented using the novel Resolution of the Coulomb Operator approach. The second evaluates electrostatic interactions between point charges, using either the smooth particle mesh Ewald method or the fast multipole method, with the latter used to simulate ion interactions in a Fourier Transform Ion Cyclotron Resonance mass spectrometer. We compare the performance of both X10 applications to state-of-the-art software packages written in other languages.

This thesis presents improvements to the X10 language and runtime libraries for managing and visualizing the data locality of parallel tasks, communication using active messages, and efficient implementation of distributed arrays. We evaluate these

improvements in the context of computational chemistry application examples.

This work demonstrates that X10 can achieve performance comparable to established programming languages when running on a single core. More importantly, X10 programs can achieve high parallel efficiency on a multithreaded architecture, given a divide-and-conquer pattern parallel tasks and appropriate use of worker-local data. For distributed memory architectures, X10 supports the use of active messages to construct local, asynchronous communication patterns which outperform global, synchronous patterns. Although point-to-point active messages may be implemented efficiently, productive application development also requires collective communications; more work is required to integrate both forms of communication in the X10 language.

The exploitation of locality is the key insight in both linear-scaling methods and the APGAS programming model; their combination represents an attractive opportunity for future co-design efforts.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction **1**
- 1.1 Scope and Problem Statement 1
- 1.2 Contributions 3
- 1.3 Thesis Outline 3

2 Programming Models and Patterns for High Performance Scientific Computing **5**
- 2.1 Distributed Memory Models 7
  - 2.1.1 MPI 8
  - 2.1.2 Charm++ 8
  - 2.1.3 Active Messages 9
  - 2.1.4 Libraries for One-Sided Communications 9
- 2.2 Shared Memory Models 10
  - 2.2.1 Pthreads 10
  - 2.2.2 OpenMP 11
  - 2.2.3 Cilk++ 12
  - 2.2.4 TBB 13
  - 2.2.5 OpenCL / CUDA 13
- 2.3 Partitioned Global Address Space Models 15
  - 2.3.1 UPC 15
  - 2.3.2 Coarray Fortran 16
  - 2.3.3 Titanium 16
  - 2.3.4 Global Arrays 17
- 2.4 Asynchronous Partitioned Global Address Space Models 17
  - 2.4.1 X10 17
    - 2.4.1.1 Active Messages in X10 20
    - 2.4.1.2 X10 Global Matrix Library 20
  - 2.4.2 Habanero Java 21
  - 2.4.3 Chapel 21
  - 2.4.4 Fortress 22

2.5 Quantum Chemistry . . . 23
2.5.1 The Self-Consistent Field Method . . . 24
2.5.2 Resolution of the Coulomb Operator . . . 26
2.6 Molecular Dynamics . . . 28
2.6.1 Calculation of Electrostatic Interactions . . . 29
2.6.2 Particle Mesh Ewald Method . . . 29
2.6.3 Fast Multipole Method . . . 31
2.6.4 Molecular Dynamics Simulation of Mass Spectrometry . . . 33
2.7 Application Patterns . . . 35
2.7.1 Dense Linear Algebra . . . 37
2.7.2 Spectral Methods . . . 37
2.7.3 N-body Methods . . . 38
2.8 Summary . . . 38

3 Improvements to the X10 Language to Support Scientific Applications 39
3.1 Task Parallelism . . . 39
3.1.1 Worker-Local Data . . . 40
3.1.1.1 Managing and Combining Worker-Local Data . . . 41
3.1.1.2 New Variable Modifiers for Productive Programming . 44
3.1.2 Visualizing Task Locality In A Work Stealing Runtime . . . 44
3.2 Active Messages . . . 46
3.2.1 Serialization of Active Messages . . . 47
3.2.1.1 Byte-Order Swapping . . . 47
3.2.1.2 Object Graphs and Identity . . . 49
3.2.2 Collective Active Messages Using **`finish/ateach`** . . . 49
3.2.2.1 A Tree-Based Implementation of **`finish/ateach`** . . . 51
3.3 Distributed Arrays . . . 54
3.3.1 Indexing of Local Data . . . 54
3.3.2 Ghost Region Updates . . . 55
3.3.2.1 Implementing Ghost Region Updates for X10 Distributed Arrays . . . 56
3.3.2.2 Evaluation of Ghost Updates . . . 59
3.4 Summary . . . 59

4 Electronic Structure Calculations Using X10 61
4.1 Implementation . . . 62
4.1.1 Parallelizing the Resolution of the Operator . . . 64
4.1.2 Auxiliary Integral Calculation with a Work Stealing Runtime . . 65
4.1.2.1 Use of Worker-Local Data to Avoid Synchronization . . 65
4.1.2.2 Overhead of Activity Management . . . 66
4.1.2.3 Optimizing Auxiliary Integral Calculations for Locality 67
4.1.3 Distributed and Replicated Data Structures . . . 68
4.1.4 Load Balancing Between Places . . . 70
4.1.5 Dense Linear Algebra Using the X10 Global Matrix Library . . . 71

4.2 Evaluation 73
4.2.1 Single-Threaded Performance 73
4.2.2 Shared-Memory Scaling 75
4.2.3 Distributed-Memory Scaling 79
4.3 Summary 82

5 Molecular Dynamics Simulation Using X10 **83**
5.1 Direct Calculation 83
5.1.1 Implementation 84
5.1.2 Evaluation 85
5.2 Particle Mesh Ewald Method 86
5.2.1 Implementation 86
5.2.1.1 Domain Decomposition With Distributed Arrays 87
5.2.1.2 Charge Interpolation Over Local Grid Points 87
5.2.1.3 Use of Ghost Region Updates to Exchange Particle Data 87
5.2.1.4 Distributed Fast Fourier Transform 89
5.2.2 Evaluation 89
5.2.2.1 Single-Threaded Performance 90
5.2.2.2 Distributed-Memory Scaling 91
5.3 Fast Multipole Method 92
5.3.1 Implementation 92
5.3.1.1 Distributed Tree Structure Using Global References . . 93
5.3.1.2 Load Balancing 95
5.3.1.3 Global Collective Operations 97
5.3.2 Evaluation 97
5.3.2.1 Single-Threaded Performance 98
5.3.2.2 Overhead of Activity Management 101
5.3.2.3 Shared-Memory Scaling 102
5.3.2.4 Distributed-Memory Scaling 103
5.4 Simulating Ion Interactions in Mass Spectrometry 106
5.4.1 Implementation 107
5.4.1.1 Integration Scheme 107
5.4.1.2 Ion Motion 108
5.4.1.3 Induced Current 108
5.4.2 Evaluation 108
5.5 Summary 111

6 Conclusion **113**
6.1 Future Work 115

Appendices **117**

A Evaluation Platforms **119**

List of Abbreviations **121**

Bibliography **123**

List of Figures

2.1 The three arms of science 5
2.2 OpenMP code for matrix multiplication $C = A \times B$ 11
2.3 Cilk code to calculate Fibonacci sequence 12
2.4 C++ code for sum over array of doubles using Intel Threading Building Blocks (TBB) 14
2.5 High-level structure of an X10 program 18
2.6 X10 code demonstrating use of closures 19
2.7 X10 code using active messages to implement a single-slot buffer 21
2.8 Chapel code demonstrating domain map and scalar promotion 22
2.9 Charge interpolation in the particle mesh Ewald method 30
2.10 Fourier Transform Ion Cyclotron Resonance Mass Spectrometer: diagram of Penning Trap 34

3.1 An improved `x10.util.WorkerLocalHandle`: key features 43
3.2 Locality of activity-worker mapping for *Histogram* benchmark 46
3.3 Locality of activity-worker mapping for *Histogram* benchmark with divide-and-conquer loop transformation 46
3.4 X10 benchmark code for serialization 48
3.5 X10 version 2.4 compiler transformation of **`ateach`** 50
3.6 Implementation of tree-based **`ateach`** in x10.lang.Runtime 51
3.7 X10 benchmark code for **`finish/ateach`** 52
3.8 Scaling with number of places of the **`ateach`** construct on *Vayu* and *Watson 4P*, and comparison with MPI broadcast. 53
3.9 Solution of a system of partial differential equations on a grid 56
3.10 Ghost region update weak scaling 60

4.1 Pseudocode for construction of Coulomb and exchange matrices using resolution of the operator 62
4.2 Variation in time to compute auxiliary integrals for different shells . . . 65
4.3 X10 code to compute auxiliary integrals, D^{lm} and B using `WorkerLocalHandle` 66
4.4 Parallel loop to compute auxiliary integrals: single activity per thread with cyclic decomposition 67

4.5 Parallel loop to compute auxiliary integrals: single activity per thread with block decomposition . . . 68
4.6 Parallel loop to compute auxiliary integrals: recursive bisection transformation . . . 69
4.7 High level structure of X10 code to compute Fock matrix . . . 70
4.8 RO contributions to K matrix: block pattern of distributed computation for different numbers of places . . . 72
4.9 Multithreaded component scaling and efficiency of RO long range energy calculation with basic parallel loop . . . 76
4.10 Locality of auxiliary integral calculation with different methods of loop division . . . 78
4.11 Multithreaded component scaling and efficiency of RO long range energy calculation with recursive bisection of loops . . . 79
4.12 Multi-place component scaling of RO long range energy calculation (8 threads per place, 3D-alanine$_4$, cc-pVQZ, $\mathcal{N}' = 11, \mathcal{L} = 19$) . . . 80

5.1 X10 code for direct calculation of electrostatic interactions between particles . . . 84
5.2 X10 code to distribute charge grid array in PME . . . 87
5.3 X10 code to interpolate charges to grid points in PME . . . 88
5.4 Subcell halo region used to calculate direct interaction in particle mesh Ewald method (PME) . . . 88
5.5 Comparison between X10 and GROMACS PME mesh evaluation on Core i7-2600 (one thread): scaling with number of particles . . . 90
5.6 Strong scaling of PME potential calculation on *Raijin* . . . 91
5.7 Low accuracy comparison between PGAS-FMM and exaFMM on Core i7-2600 (one thread): scaling with number of particles . . . 98
5.8 Time for M2L transformation for different orders of expansion p . . . 100
5.9 Higher accuracy comparison between PGAS-FMM and exaFMM on Core i7-2600 (one thread): scaling with number of particles . . . 101
5.10 Multithreaded component scaling and efficiency of PGAS-FMM on Core i7-2600 . . . 103
5.11 Locality of activity-worker mapping for FMM force evaluation . . . 104
5.12 Strong scaling of FMM force calculation on *Raijin* . . . 104
5.13 Strong scaling of FMM force calculation on *Watson 2Q* . . . 105
5.14 MPI communication map for FMM force calculation on *Raijin* . . . 106
5.15 FTICR-MS ion cloud evolution in first 2.5 ms of simulation: packet of 5000 lysine and 5000 glutamine ions. . . . 110

List of Tables

2.1 The 'thirteen dwarfs': patterns of communication and communication . 36

4.1 Time to compute Fock matrix for different molecules using RO 74
4.2 Multithreaded component scaling of RO long range energy calculation on *Raijin* with different methods of dividing integral and J matrix loops between worker threads (8 threads, 3D-alanine$_4$, cc-pVQZ, $\mathcal{N}' = 11, \mathcal{L} = 19$) 77
4.3 Distributed Fock matrix construction using resolution of the Coulomb operator (RO): load imbalance between places 80
4.4 Distributed K matrix construction using RO: floating-point and communication intensity 81

5.1 Direct calculation of electrostatic force: cycles per interaction, X10 vs. C++. 86
5.2 Component timings of PGAS-FMM for varying numbers of particles and accuracies 99
5.3 Slowdown due to X10 activity management overhead for PGAS-FMM . 102
5.4 Amino acids in ANU mass spectrometer: timings on *Raijin* 109
5.5 Amino acids in ANU mass spectrometer: predicted and measured frequencies 110

Chapter 1

Introduction

This thesis concerns the interplay between programming models and scientific applications. Specifically, it considers the asynchronous partitioned global address space programming model and its use in the productive development of high performance scientific application codes.

1.1 Scope and Problem Statement

Applications of scientific computing, in particular physical simulations, tend to exhibit properties of *scale* and *physical locality*. Scale means a system may be simulated at different levels of accuracy depending on the phenomena of interest. Physical locality means that interactions between events or entities that are nearby in space or time are stronger than those between distant events. *Linear-scaling methods* exploit physical locality by approximating distant interactions, so as to reduce the computational complexity of simulation. These methods generate irregular parallelism, where the amount of computation required for each given portion of the system is different depending on its importance to the simulation as a whole.

When implementing a scientific algorithm for execution on a high performance computing architecture, the primary concerns of parallel computing are:

- *Parallelism*: the computation must be divided into tasks to be executed in parallel, producing an even load between processing elements, with minimal task overhead.
- *Data locality*: data must be arranged in memory to minimize the amount of movement between levels in the memory hierarchy and between distributed processing elements (communication).
- *Synchronization*: the computation must be ordered for correctness with minimal locking or other wait overhead.

The properties of scientific applications have a direct bearing on these concerns: physical scale tends to generate irregular parallelism in simulation, and physical locality

has a direct bearing on data locality and synchronization between computational tasks.

Traditional programming languages such as C++ and Fortran fail to address the concerns outlined above. To fill the gap, programmers have used extensions based on either a shared memory model such as Open Multi-Processing (OpenMP), or a distributed memory model such as the Message Passing Interface (MPI). As almost all modern high performance computing systems have multiple levels of parallelism, with different levels corresponding to both memory models, the dominant programming model is now a hybrid of a sequential language, OpenMP and MPI. As well as effectively requiring programmers to learn three different programming languages, the hybrid model results in several conceptual mismatches. For example, synchronization between MPI processes is achieved through matching message calls, whereas OpenMP uses explicit synchronization constructs such as barriers and atomic directives; and data movement between memory spaces in MPI is explicit in communication, whereas OpenMP does not explicitly recognize data movement between areas of the memory hierarchy. The ease with which the programming model supports the expression of parallelism has been identified as a dominating factor in programming productivity for high performance computing (HPC) [Kuck, 2004].

In comparison with established HPC programming models, the asynchronous partitioned global address space model [Saraswat et al., 2010] has two distinguishing features: distributed task parallelism and a global view of memory which accounts for data locality. The aim of this research has been to investigate whether this model, as realized in the X10 programming language[1], allows the *succinct expression* of typical scientific application codes while achieving *performance* comparable to or better than established programming languages and libraries. The programming model, however, is only one factor in the clarity and performance of a given application code. Of arguably equal importance are *data structures* and *high level operations* provided by the runtime or third-party libraries. These are the building blocks from which the most important parts of any application are constructed, and allow the programmer to operate at a higher level of abstraction than individual numerical or memory update operations. Finding the right data structures and operations to express a given algorithm can greatly reduce both the development and execution time of the application.

This thesis seeks to address the following questions:

- How can the explicit representation of data locality and task parallelism in the APGAS programming model be used to succinctly express data structures and high-level operations that support scientific applications?
- How does the performance of these operations compare with traditional models? What are the barriers to performance, and how can these be overcome?

These questions are considered with regard to the three concerns of parallel computing identified above.

[1] http://x10-lang.org

Unavoidably, there are many important aspects of high performance scientific computing that are not addressed in this thesis. For example, development productivity is a major factor in 'time to solution' and was the key driver for the Defence Advanced Research Projects Agency (DARPA)'s High Productivity Computing Systems project which funded the initial development of the X10 language [Dongarra et al., 2008]. Objective measurement of development productivity is a complex systems analysis problem, therefore productivity is not considered in this thesis except insofar as 'clarity' or 'expressiveness' may be taken as subjective factors in total development effort. Another key concern for high performance computing is fault tolerance, as increasing system complexity reduces mean time between failures. Finally, numerical accuracy is a perennial concern; the correctness of results, as well as the balance between accuracy and computation time, are critical to the effective application of scientific computing. Fault tolerance and numerical accuracy are both areas in which the programming model may be expected to play a role, but they are orthogonal to the concerns addressed in this thesis.

1.2 Contributions

This thesis presents results from an ongoing co-design effort involving researchers at IBM and ANU, in which a new programming model – the asynchronous partitioned global address space model – is applied to high performance scientific computing. Insights gained from application development inform language design and vice versa.

The contributions presented in this thesis fall into two main categories: firstly, enhancements to the X10 programming language or the APGAS programming model to improve expressiveness or performance, and secondly, the development of scientific application codes, high level data structures and operations to motivate and evaluate such improvements. Many of the concepts and techniques used in the X10 programming language are found in other languages, and as such this work has a wider application than the X10 language itself.

1.3 Thesis Outline

The presentation of this thesis is focused on two example applications from the field of computational chemistry: molecular dynamics and the Hartree–Fock method; this reflects an early decision that this research should be strongly application-driven. These applications were the first significant X10 codes created outside of the X10 development team, and as such represented an important source of external feedback for the language designers.

Chapter 2 presents an overview of programming models used in high performance scientific computing, and presents the background for the scientific application examples discussed in later chapters. Chapter 3 expands on key features of the X10 programming language and the APGAS programming model, and proposes improvements to the X10 language and runtime libraries to better support scientific

programming. Chapters 4 and 5 describe the application of the APGAS programming model to two different scientific problems: the implementation of a Hartree–Fock self-consistent field method, and calculation of electrostatic interactions in a molecular dynamics simulation. Finally, Chapter 6 summarizes how the contributions presented in this thesis support productive programming of high performance scientific applications, and identifies opportunities for future research.

In evaluating the contributions, we conducted performance measurements on five parallel machines, ranging from a typical desktop machine to a highly-multithreaded, tightly integrated compute cluster; these machines are described in detail in appendix A.

All application codes described in this thesis are available at http://cs.anu.edu.au/~Josh.Milthorpe/anuchem.html and are free software under the Eclipse Public License.

Chapter 2

Programming Models and Patterns for High Performance Scientific Computing

The goal of science is to increase our understanding of the physical world, to explain and predict natural phenomena. Traditionally, the two arms of science have been modelling and experiment. Scientists construct a simplified theoretical *model* of physical phenomena; this is tested and refined based on observations of the real world gained through *experiment*.

Computing has enabled the development of a third arm of modern science: *simulation*. By translating abstract models into numerical simulations, scientists can compare systems with different parameters and starting conditions, observe simulated phenomena at scales that may be impossible to observe in a real-world experiment, and quickly test whether a change in the model improves its correspondence to experimental data. Figure 2.1 shows the relationship between these three arms of modern science: simulation is used to understand and refine a model, which can then be tested against experimental observations of the natural world.

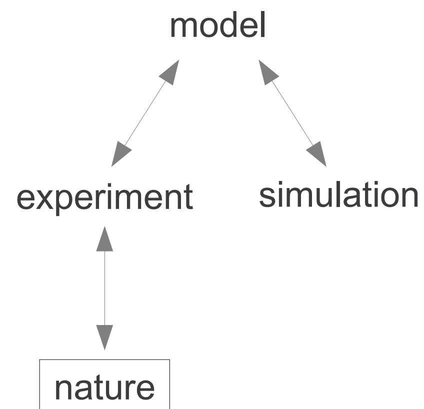


Figure 2.1: The three arms of science

In the same way that scientists use simplified models to understand physical phenomena, computer programmers construct simplified models of computer architectures. Modern computer architectures are unavoidably complex. Typical descriptions of modern architectures run into hundreds or thousands of pages (for example, Intel [2013a]; IBM [2012]), describing instruction sets, functional units and pipelines, memory management, cache hierarchies and I/O. A *programming model* provides a framework

which captures those features of the architecture that are essential to the programmer while abstracting away less important details. A good programming model enhances programmer productivity and may be applied to a whole class of similar architectures.

Typical scientific models exhibit three properties which have particular bearing on computer simulation:

1. Measure: Descriptions of physical phenomena are quantitative, meaning that simulation requires numerical computing.

2. Locality: Interactions between events or entities that are nearby in space or time are stronger than those between distant events.

3. Scale: A system exhibits qualitatively different behavior depending on the spatial or temporal scale at which it is observed, and may therefore be simulated at different levels of accuracy depending on the phenomena of interest. (Notable exceptions to this rule exist in non-linear and chaotic systems.)

The latter two properties are of particular interest when considering the implementation of the physical model as a computer code:

- Locality: Physical locality in the scientific model may be mapped to data locality in the simulation. Computer systems also exhibit locality, in the form of a distance-dependent cost for data transfers between portions of the memory system (e.g. between levels in the cache hierarchy or between local and remote memory spaces). As memory access costs become increasingly non-uniform, a programming model which makes these costs explicit would support greater performance portability and transparency.

- Scale: The requirement to simulate areas of greater interest at higher levels of accuracy translates into irregular computation. This presents a challenge for load balancing, which is becoming more important with the trend towards larger numbers of hardware threads. An *asynchronous*, task-based programming model is an attractive approach for load balancing irregular computations on highly multithreaded architectures.

This thesis will argue that the asynchronous partitioned global address space (APGAS) programming model is well suited to implementation of scientific applications on modern architectures, as it combines the explicit representation of data locality with task-based parallelism.

The two parallel programming models most commonly used in high performance computing are the *distributed memory* and *shared memory* models. In the distributed memory model, the memory space for each process is private; whereas in the shared memory model, the entire memory space is shared between all processes. The emerging *partitioned global address space (PGAS)* model combines features of shared and distributed memory, allowing all processes to access memory anywhere in the system while still accounting for locality. X10 and related APGAS programming languages

extend the PGAS model with support for *asynchronous tasks*. Sections 2.1–2.4 discuss the key features of each model and some commonly used HPC programming technologies which exemplify them.

A physical simulation may describe the target system at different levels of approximation. At the highest level of description, systems of particles might be modeled according to relativistic quantum mechanics. Successive approximations may be made to create a hierarchy of models; lower level (coarser grained) models lose the ability to model certain kinds of behavior, but gain applicability to larger systems or longer timescales [Berendsen, 2007, §1.2]. This thesis presents applications from the domain of computational chemistry at two different levels of the hierarchy. Quantum chemistry, introduced in section 2.5, models the interactions between atomic nuclei and electrons in bound states. Molecular dynamics, introduced in section 2.6, operates at a higher level of abstraction and approximates the interactions between atoms and molecules using classical mechanics. While both of these levels are relatively high on the hierarchy of simulation (compared to, for example, fluid dynamics), they represent significant classes of scientific application programs, including the bulk of materials science applications. They are therefore of interest in evaluating programming models for current and emerging architectures.

Finally, section 2.7 places the application examples used in this thesis within broad classes of scientific and engineering applications. Such a classification is useful in programming language–application co-design as it helps to determine the scope and generality of design choices.

2.1 Distributed Memory Models

In the distributed memory model, computation is divided between a number of processes, each with its own private address space. Processes communicate via explicit messages, which may be passive (data transfer only) or active (also initiating computation at the receiving end). Two-sided communications require the cooperation of sender and receiver, whereas one-sided communications allow one process to directly access data owned by another process without its explicit cooperation.

The most widely used distributed memory model is the process model, first defined for Unix [Ritchie and Thompson, 1974] and now ubiquitous in parallel operating systems. Each process may execute a different program; processes communicate by means of pipes or sockets. The Unix process model, while efficient and flexible, is at too low a level for productive programming of portable, high performance applications. The programmer must manage message formats, buffering and synchronization. Furthermore, pipes and sockets are difficult to combine with hardware support for efficient network operations; for example, row broadcast on a torus network. To insulate the programmer from such low-level details, and to enable portability to a wide range of parallel architectures, higher-level programming models are needed. The following subsections describe approaches to distributed memory programming based on two-sided messaging (MPI), remote procedure calls (Charm++), low-level active messages, and libraries for one-sided communications.

2.1.1 MPI

The Message Passing Interface (MPI) was developed to address the issues of programmability mentioned above with the first standard, MPI-1, released in 1994 [MPI Forum, 1994]. MPI defines a set of high-level message functions along with data types, collective operations and operations for creating and managing groups of processes. Originally, all message functions were two-sided operations in which each call from a sending process must be matched with a corresponding call from the receiving process. MPI-2 added one-sided communications such as *put* and *get*, which have the potential to map to hardware support for remote direct memory access (RDMA) available on modern architectures [MPI Forum, 2003].

The availability of implementations of MPI for all high performance architectures makes it an attractive candidate as a communications layer on top of which to implement high-level languages. However, the MPI-2 model of one-sided communications received some criticism that it was a poor match for new partitioned global address space parallel languages [Bonachea and Duell, 2004]. Partly in response to these criticisms, MPI-3 implemented a new memory model for one-sided communications, as well as adding non-blocking and sparse collective operations [MPI Forum, 2012].

MPI has been exceedingly successful in high performance computing, such that it may be considered as the de facto standard for distributed memory programming. There are high performance implementations of MPI available for every major HPC architecture, and MPI has been used to implement a vast range of scientific applications. Despite these successes, MPI has a number of drawbacks from the point of view of programmer productivity. Firstly, the message passing model encourages a fragmented view of computation, in which data structures and control flow are explicitly decomposed into per-process chunks [Chamberlain et al., 2007]. The details of decomposition tend to obscure the high-level features of the computation. Secondly, MPI processes do not share data except by explicit communication; in contrast, modern computing architectures are typically composed of multiple cores with a shared cache hierarchy. To take full advantage of such systems, the programming model must reflect the sharing of data between tasks. Section 2.2 describes such shared memory models.

2.1.2 Charm++

Charm++ is an extension to C++ for message-driven parallel programming [Kalé and Krishnan, 1993]. It is based on migratable objects called *chares*, which interact through asynchronous method invocations. New chares can be created by a *seed* — a constructor invocation which may be called on any processing element to instantiate the chare. The ability to migrate chares allows the runtime system to provide fault tolerance and adapt to changing processor resources. The Charm++ runtime system has been used to implement Adaptive MPI, which provides dynamic load balancing and multithreading in a traditional MPI framework [Huang et al., 2003].

Charm++ uses shared memory to optimize communication between processing elements located on the same node. However, it does not otherwise take advantage

of shared memory by, for example, allowing multiple chares to operate on shared data structures.

2.1.3 Active Messages

Active messages are an alternative to two-sided messages, which allow overlapping of computation and communication with minimal overhead [von Eicken et al., 1992]. They differ from general remote procedure call mechanisms in that an active message handler is not required to perform computation on data, but merely to extract data from the network and integrate them into ongoing computation. An active message contains the address of a user-level handler to be executed on receipt of the message.

Active messages are a primitive communication mechanism, which may be used to implement higher-level communication models including message passing. For example, the Parallel Active Message Interface (PAMI) is used on the Blue Gene/Q supercomputer to implement MPI [Kumar et al., 2012].

2.1.4 Libraries for One-Sided Communications

While efforts to define a standard for two-sided communications achieved success through MPI, approaches to one-sided communications have been more varied. During the 1990s, vendors developed platform-specific interfaces for one-sided communications (or remote memory access) such as Cray's SHMEM [Barriuso and Knies, 1994] and IBM's LAPI [Shah et al., 1998]. Following these efforts, the Aggregate Remote Memory Copy Interface (ARMCI) was created as a portable library for efficient one-sided communications [Nieplocha et al., 2006b]. ARMCI has been used to support the global address space communication model implemented in the Global Arrays library (see §2.3.4), complementing message-passing libraries like MPI, and has also been used to implement Coarray Fortran (CAF) (see §2.3.2). Remote memory operations in ARMCI do not require any action by the receiving process in order to progress. ARMCI supports non-contiguous data transfers, which are important in scientific applications involving multidimensional arrays. It also provides remote atomic accumulate and read-modify-write operations and distributed mutexes.

While ARMCI provides powerful communication primitives, it is targeted at the development of runtime systems and libraries rather than at application programming [Nieplocha et al., 2006b]. The lack of explicit rules for synchronization requires some care to avoid errors in using ARMCI remote put operations, and may lead to complications when combined with sequential programming models.

GASNet is a portable library of one-sided communication primitives designed for implementing PGAS languages [Bonachea, 2002]. Its core application programming interface (API) is based on active messages; it also provides an extended API featuring put and get operations that may take advantage of hardware support for RDMA. GASNet has been used to implement single program, multiple data (SPMD) PGAS languages like CAF, Unified Parallel C (UPC) and Titanium (see §2.3) as well as Chapel (see §2.4.3).

By themselves, active messages and libraries for one-sided communication do not constitute a parallel programming model; a complete model must also define how the computation is to be divided locally into parallel tasks, and how tasks and messages should synchronize.

2.2 Shared Memory Models

In the shared memory model, a single address space is shared between a number of threads, each of which has a private stack and control flow. 'Communication' is implicit, via threads operating on shared data structures and observing the results of updates from other threads. To ensure consistency of operations, explicit synchronization between threads is required by means of memory barriers, locks or higher-level synchronization constructs. Shared memory models may be further divided into threading models, in which the programmer divides the computation into units of work according to available hardware parallelism; and task-based models, in which the programmer exposes potentially parallel tasks to be scheduled by a runtime system for execution on one or more processing elements. The following sections describe a pure threading model (Pthreads), an API which supports both threading and task-based programming (OpenMP), two purely task-based models (Cilk and Intel Threading Building Blocks (TBB)), and finally the OpenCL and CUDA models for accelerators, which address issues of partitioned memory spaces arising from programming for accelerators such as GPUs.

2.2.1 Pthreads

The most widely used example of the shared memory model is the Posix Threads standard (Pthreads) [IEEE, 2008], which defines an API for creating and managing threads and is implemented for all UNIX-like systems. Threads synchronize through barriers, mutexes and condition variables.

The Pthreads model maps closely to underlying features of typical operating systems and hardware; it is therefore an efficient programming model for high performance computing. The use of Pthreads requires, however, significant 'boilerplate' code to create and synchronize threads, which is a barrier to achieving high productivity for application programming. Moreover, the explicit threading model implies an undesirable tradeoff in terms of load balancing: executing with a minimal number of threads may increase load imbalance (the time spent waiting for idle threads), whereas over-subscribing threads creates additional overheads in the operating system and scheduler. Lighter-weight threading models such as Nano-threads [Martorell et al., 1996], Qthreads [Wheeler et al., 2008] and MassiveThreads [Nakashima and Taura, 2014] reduce the overhead of creating threads, however, a more promising approach is to decouple the units of execution from the executing threads altogether. This is the approach used in shared memory *task-based* model implemented in high-level APIs such as OpenMP, Cilk and TBB.

```
1  double *a = new double[M*K]();
2  double *b = new double[K*N]();
3  double *c = new double[M*N]();
4  #pragma omp parallel for schedule(static) collapse(2)
5  for (size_t j = 0; j < N; j++) {
6      for (size_t i = 0; i < M; i++) {
7          double x = 0.0;
8          for (size_t l = 0; l < K; l++) {
9              x += a[i+l*K] * b[l+j*N];
10         }
11         c[i+j*N] = x;
12     }
13 }
```

Figure 2.2: OpenMP code for matrix multiplication $C = A \times B$

2.2.2 OpenMP

OpenMP [Dagum and Menon, 1998] is a portable, high-level API for shared memory parallel programming in C, C++, and Fortran. It supplements these languages with compiler directives, runtime library functions and environment variables. OpenMP uses a fork-join model like that of Pthreads, with code executing sequentially until it enters a *parallel region*. Work-sharing directives support both data parallelism (over loops) and functional decomposition (using parallel *sections*).

Figure 2.2 shows an OpenMP code to perform the matrix multiplication $C = A \times B$. The compiler directive on line 4 defines a parallel region over the following block (lines 5-12); it also declares that the following `for` loop(s) should be executed in parallel. The `schedule(static)` clause declares that loop iterations should be divided evenly between threads using a static block division; `collapse(2)` means that the iteration space of the following two loops (over `M` and `N`) should be divided in two dimensions between all threads.

OpenMP programs are structured in terms of teams of cooperating threads. The primary model of parallelism in OpenMP is explicit threading: parallel sections create a defined number of threads, between which work can be shared. Explicit threading makes it difficult to compose parallelism at multiple levels using nesting due to a combinatorial explosion in the number of threads [McCool et al., 2012, chapter 1]. For this reason, the default usage is to disable nested thread creation.

OpenMP 3.0 [OpenMP ARB, 2008] introduced parallelism using tasks, which may be executed by a work stealing scheduler [Ayguade et al., 2009]. Tasks allow the convenient representation of divide-and-conquer and irregular applications. However, OpenMP tasks may not include nested work-sharing constructs, which is a limitation when calling parallel library code [Teruel et al., 2013]. This is in contrast to the task-based shared memory programming models of Cilk and TBB, which support arbitrary nesting. The task-based approach also generalizes readily to multiple levels of parallelism on distributed memory systems.

```
1  cilk int fib(int n) {
2      if (n < 2) return n;
3      else {
4          int n1, n2;
5          n1 = spawn fib(n1);
6          n2 = fib(n2);
7          sync;
8          return (n1 + n2);
9      }
10 }
```

Figure 2.3: Cilk code to calculate Fibonacci sequence

OpenMP 4.0 [OpenMP ARB, 2013] added support for accelerators such as graphics processing units (GPUs) using `target` constructs to manage data and tasks on the target device, and *places* to define thread affinity policies to improve data locality. In these respects OpenMP is moving towards explicit representation of data locality similar to PGAS models (see §2.3).

OpenMP has enjoyed significant success due to widespread compiler support as well as the ease with which programmers can incrementally add parallelism to existing codes using compiler directives. The most common use of OpenMP for HPC is to express shared-memory parallelism within a larger distributed memory code, e.g. intra-node parallelism for each node of a cluster. This approach requires the programmer to decompose the program at two levels, leading to a fragmented view of the algorithm.

2.2.3 Cilk++

Cilk++ [Blumofe and Leiserson, 1999; Leiserson, 2010] is an extension to C/C++ for task parallelism using a fork-join model. The programmer converts a normal serial program to a parallel program by annotating with special Cilk keywords to fork and join tasks. A function is marked for possible execution in parallel using the keyword `cilk`; it may then be called as a separate task using the keyword `spawn`. The keyword `sync` is used to wait for termination of all tasks started by the current block. Figure 2.3 demonstrates the use of these keywords in a Cilk program to compute the Fibonacci sequence in parallel.

The Cilk runtime was the first to schedule tasks through *work stealing*. The runtime maintains a pool of worker threads which execute activities, each with its own double ended queue or *deque*[1] of tasks. When a worker encounters a `spawn` statement, it pushes a task to the top of its own deque. If another worker thread is idle, it picks another worker at random and attempts to steal a task from the bottom of its deque. Thus the victim and the thief operate on opposite ends of the deque, reducing the need for synchronization.

[1] pronounced 'deck'

Recognizing the importance of data parallelism and loop structures to typical applications, Cilk provides a keyword `cilk_for`, which allows the iterations of a loop to run in parallel [Leiserson, 2010]. The loop is automatically parallelized using divide-and-conquer recursion. Divide-and-conquer parallelism has positive implications for data locality in some loops, which will be discussed in chapter 3.

Cilk-style work stealing has been adopted in numerous programming languages and libraries, including the Java Fork-Join framework [Lea, 2000], X10 (see §2.4.1), Chapel (§2.4.3) and TBB.

2.2.4 TBB

Intel Threading Building Blocks (TBB) [Reinders, 2010] is a C++ template library that supports task parallelism. It focuses on divide-and-conquer parallel algorithms and thread-safe data structures, and it implements load balancing using Cilk-style work stealing.

The TBB scheduler, like other work-stealing schedulers, requires the problem to be divided into a set of nested tasks. To this end, it provides for parallel loops to be created by recursively subdividing a loop range. TBB introduces the concept of a *splittable type*, which defines a splitting constructor that allows an instance of the type to be split into two pieces.

By using standard C++ template programming practices, TBB provides an accessible path for C++ programmers to incrementally add parallelism to existing applications. For example, the code in figure 2.4(a) calculates a simple sum reduction over an array of double-precision floating-point numbers. Figure 2.4(b) shows how TBB's `parallel_loop` construct can be used to divide this loop into a set of tasks to be executed in parallel. While the loop-splitting approach is powerful, defining loop body objects for each parallel loop using the C++98 standard requires rather verbose code. The use of lambda functions introduced in the C++11 standard reduces this burden [Robison et al., 2008]. Figure 2.4(c) shows code for the same parallel sum using TBB with lambda expressions.

TBB directly supports a number of commonly used parallel programming patterns including map, reduce, pipeline and task graph. It also provides multiple flavors of mutex and atomic operations to synchronize between tasks, and scalable memory allocation routines designed for use by multiple threads.

The direct support for fundamental parallel patterns means that many parallel applications may be expressed concisely using TBB. It assumes a shared-memory model, meaning that a complementary distributed memory model such as MPI is required to address issues of data locality.

2.2.5 OpenCL / CUDA

The Open Computing Language (OpenCL) framework [Khronos, 2012] supports the execution of compute kernels on accelerators such as GPUs, digital signal processors or attached co-processors like the Intel Xeon Phi. The kernels are specified in an extension of C99 which allows a computation to be divided into a regular grid of

```
1  double SerialSum(double a[], size_t n) {
2      double sum = 0.0;
3      for (size_t i = 0; i < N; i++) {
4          serial_sum += a[i];
5      }
6      return sum;
7  }
```

(a) Serial sum

```
1  struct Sum {
2      double value;
3      Sum() : value(0.0) { }
4      Sum(Sum& s, tbb::split) { value = 0.0; }
5      void operator()(const tbb::blocked_range<double*>& r) {
6          double res = value;
7          for (double* v = r.begin(); v != r.end(); v++) {
8              res += *v;
9          }
10         value = res;
11     }
12     void join(Sum& rhs) { value += rhs.value; }
13 };
14
15 Sum sum_body;
16 tbb::parallel_reduce(tbb::blocked_range<double*>(a, a+N), sum_body);
17 double sum = sum_body.value;
```

(b) Parallel sum using TBB

```
1  double sum2 = tbb::parallel_reduce(
2      tbb::blocked_range<double*>(a, a+n),
3      0.0,
4      [](const tbb::blocked_range<double*>& r, double value)->double
5      {
6          return std::accumulate(r.begin(), r.end(), value);
7      },
8      std::plus<double>()
9  );
```

(c) Parallel sum using TBB and lambda expressions

Figure 2.4: C++ code for sum over array of doubles using TBB

work items, which are processed in work groups (mirroring the architecture of a typical GPU). There are mechanisms for allocating and sharing memory between threads in a work group, and synchronizing within a group. Kernels are executed asynchronously on the accelerator through a queuing system, and may also be executed on a standard CPU. The OpenCL memory model distinguishes between the main memory of the host device and memory of the accelerator device, and an API is provided to manage transfers between the two. Device memory is further divided into private memory, local memory accessible by all threads in a work group, constant memory that is readable by all threads, and global memory. The Compute Unified Device Architecture (CUDA) framework [NVIDIA, 2013] is similar to OpenCL, but is specific to NVIDIA GPUs.

The OpenCL and CUDA programming models reflect the divisions in memory of typical GPU architectures, and the cost of transferring between portions of the memory. However, they also allow for threads to share memory using a relaxed memory consistency model. As such they combine elements of both distributed and shared memory models. The close mapping between these models and target architectures (GPUs) supports the development of high performance codes, however, it may not be possible to achieve performance portability to other non-GPU architectures.

2.3 Partitioned Global Address Space Models

The partitioned global address space (PGAS) model is similar to the shared memory programming model in that all threads of execution have access to a global shared memory space. However, in the PGAS model, the memory space is divided into local partitions, with an implied cost to moving data between partitions. The programmer has control over the placement of data and consequently computation over those data, which is critical for high performance on modern computers [Yelick et al., 2007a].

2.3.1 UPC

A reliable approach to implementing the PGAS model is to extend an existing programming language. UPC [UPC Consortium, 2005] extends ANSI C with `shared` (global) objects. Shared objects may be divided into portions local to each thread in the computation, but each thread may access remote data using shared pointers. UPC also defines a user-controlled memory consistency model, in which each memory access is either *strict* in the sense of sequential consistency [Lamport, 1979] or *relaxed* in which case ordering of memory accesses is only preserved from the point of view of the local thread. UPC also introduced a *split-phase barrier*, in which threads signal arrival at the barrier and may then continue to do useful (purely local) work until all other threads have arrived at the barrier. This may be used to reduce idle time due to barrier synchronization, in applications where the work may be divided into distributed and local portions. UPC also provides a number of collective 'relocalization' functions (broadcast, scatter, exchange) and computation functions (reduce, prefix reduce) similar to those defined by MPI.

UPC's simple locality model provides performance portability across a range of shared and distributed memory systems. SPMD-style applications written in UPC have achieved high performance on the largest computing clusters. It is less well suited to irregular applications requiring load balancing, as data decompositions are static and new threads cannot be created. Min et al. [2011] propose a dynamic tasking library and API as an extension to UPC to support such applications.

2.3.2 Coarray Fortran

CAF is an extension of Fortran for SPMD programming with the partitioned global address space model [Numrich and Reid, 1998; Mellor-Crummey et al., 2009]. Multiple process images execute the same program, each with its own local data. The key concept in CAF is the *coarray*, which is an array shared between multiple images in which each image has a local portion of the array, but may directly access data local to other images. The original coarray extensions (which have since been adopted into the Fortran 2008 standard) required coarrays to be statically allocated across all processes. Later work [Mellor-Crummey et al., 2009] expands CAF to support dynamically allocated arrays over subsets of images, and global pointers for the creation of general distributed data structures.

By extending Fortran, CAF builds on decades of effort in the development of high-performance compilers and application codes, and provides an evolutionary pathway for existing codes to exploit parallelism using the PGAS model.

2.3.3 Titanium

Titanium is a parallel dialect of Java designed for high performance scientific computing [Yelick et al., 1998]. Titanium extends serial Java with multi-dimensional arrays, separating the index space (domain) from the underlying data, and an unordered loop construct, `foreach`, which allows arbitrary reordering of loop iterations to support optimizations such as cache blocking and tiling [Yelick et al., 2007b]. To avoid the overheads associated with boxed types in Java, Titanium supports the definition of *immutable* classes, which save memory and preserve locality by dispensing with pointers.

Titanium follows the SPMD model of parallelism; processes synchronize at barrier statements and a single-qualification analysis is used to ensure that all processes encounter the same sequence of barriers. Distributed data structures such as distributed arrays may be constructed using *global pointers*, which may refer to objects in the local partition or a remote partition. Difficulties in implementing full distributed garbage collection motivated the introduction of memory *regions*, into which objects may be allocated and an entire region de-allocated with a single method call [Yelick et al., 2007b]. (A better solution to this problem was subsequently implemented for the X10 language; see 2.4.1.)

2.3.4 Global Arrays

The Global Arrays library [Nieplocha et al., 2006a] provides PGAS support for array data, and has been used successfully as a component in computational chemistry codes such as NWChem [Valiev et al., 2010] as well as a range of other codes. Global Arrays supports one-sided put and get operations, synchronization and collective operations but does not support active messages.

2.4 Asynchronous Partitioned Global Address Space Models

The asynchronous partitioned global address space (APGAS) model extends the PGAS model with support for asynchronous tasks. APGAS is exemplified by three languages developed under the DARPA High Productivity Computing Systems (HPCS) program [Dongarra et al., 2008], which aimed to reduce 'time to solution' for high performance applications. It has been widely acknowledged that the non-uniform programming models prevalent in HPC are a limitation on programmer productivity [Hochstein et al., 2005; Dongarra et al., 2011]. Accordingly, one component of the program was the development of new parallel programming models and languages. Initially, three companies were funded to develop new programming languages:

- IBM - the *X10* language;
- Cray - the *Chapel* language; and
- Sun - the *Fortress* language.

DARPA funding for Fortress was discontinued in 2007 and active development on the project ended in 2012. Although DARPA funding for X10 and Chapel ended in 2012 with the completion of the HPCS program, both IBM and Cray continue active development of their languages.

All three HPCS languages use the APGAS programming model. The APGAS model provides a global view of memory in which a portion of the memory is local to each process. It is therefore straightforward to write programs in which processes access remote data, while still accounting for locality and associated communication costs. X10 is entirely explicit about locality whereas Chapel and Fortress support operations with implicit remote access. They are all strongly-typed languages supporting object-oriented programming with parameterized types.

2.4.1 X10

X10 [Charles et al., 2005] is an APGAS language which explicitly represents locality in the form of *places*. The sequential core of X10 borrows much from Java, and is fully interoperable with it [Saraswat et al., 2014]. To this core, X10 adds a small number of powerful parallel constructs and rules for their combination. A *place* in X10 corresponds to one or more co-located processing elements with attached

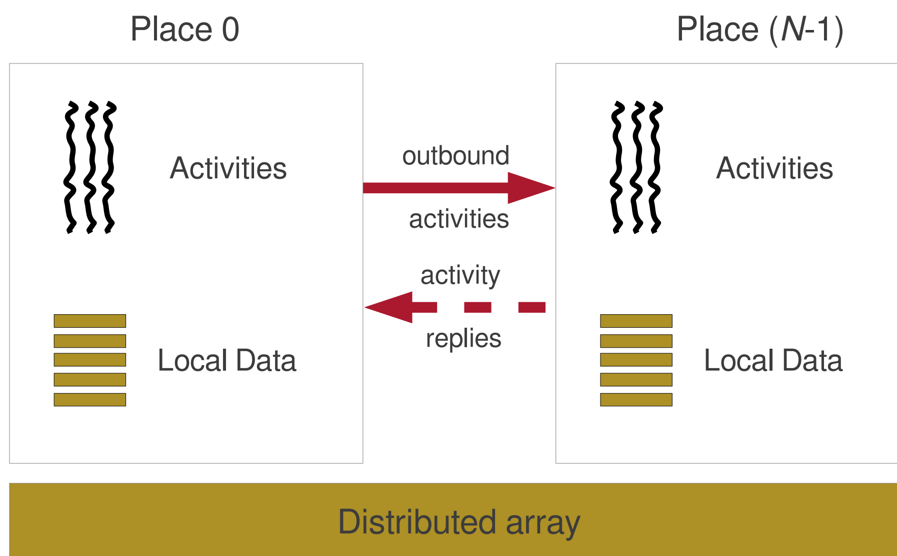


Figure 2.5: High-level structure of an X10 program

local storage. The **`async`** statement provides *task parallelism*, allowing the creation of activities at any place, which may be synchronized using **`atomic`** blocks. The **`at`** statement combines communication and computation in *active messages*, and the **`finish`** construct supports distributed termination detection. Figure 2.5 shows the high-level structure of an X10 program.

Mutual exclusion is ensured between activities running at the same place by the use of conditional and unconditional atomic blocks (**`when`** and **`atomic`**). An atomic block executes as though all other activities in that place are suspended for the duration of the block. A conditional atomic block `when(c)S` suspends until such time as the condition `c` is true and then executes the statement `S` atomically. Correctness requires that any update by another activity affecting the condition `c` must also be made within an atomic block; otherwise it would be invisible to the activity waiting on the condition. Activities can be **`clocked`** – synchronized in phased computation, where all activities must advance to the next phase together.

X10 also provides a rich array sub-language, based in part on the ZPL array language [Chamberlain, 2001]. An *array* is a collection of objects which are indexed by *points* in an arbitrary bounded *region*. A key data structure in X10 is the *distributed array*. Each element in a distributed array is assigned to a particular place according to the array *distribution*, which is a mapping from point to place. Distributed arrays are very general in scope: they allow for arbitrary distributions (for example, block, block-cyclic, recursive bisection, fractal curve) and arbitrary regions (for example dense or sparse, rectangular, polyhedral or irregular). X10 is also designed using the multi-resolution approach, so the array library and runtime are largely implemented using lower-level language constructs of X10.

X10 supports first class functions called *closures* (in the tradition of the Scala

language). The X10 code example in figure 2.6 demonstrates the use of a first class function to evaluate an expectation value for a two-particle property of a molecular ensemble. The closure `radialDistribution()` (lines 17-18) is defined to calculate

```
1  public class MolecularEnsemble {
2    val mols:Array[Molecule](1);
3    ...
4    public def expectationValue(
5        twoParticleFunction:(a:Molecule,b:Molecule) => Double
6    ):Double {
7          var total:Double = 0.0;
8          for ([i] in mols)
9              for ([j] in mols)
10                 if (i != j) {
11                     total += twoParticleFunction(mols(i), mols(j));
12                 }
13         val N = mols.size;
14         return total / (N*(N-1));
15     }
16 }
17 val radialDistribution =
18     (a:Molecule,b:Molecule) => b.center.distanceFrom(a.center);
19 val ensemble:MolecularEnsemble = ...;
20 val mu = ensemble.expectationValue(radialDistribution);
```

Figure 2.6: X10 code demonstrating the use of a first class function to evaluate the expectation value of the two-particle function `radialDistribution()` for a molecular ensemble

the distance between two molecules. It is passed to the method `expectationValue()` (lines 4–16), which executes the closure over all pairs of molecules in the system and returns the average (expectation) value. The same `expectationValue` method could be used to calculate any two-particle property of the ensemble, simply by passing it a different closure.

X10 addresses the issue of load balancing through work-first work stealing between worker threads at each place [Tardieu et al., 2012]. The X10 *global load balancing framework* [Saraswat et al., 2011] extends this approach to work stealing between places using lifeline graphs for distributed termination detection. The local (shared memory) and global (distributed memory) work stealing implementations have not yet been integrated.

X10 provides a global reference type `GlobalRef`, and a `PlaceLocalHandle` type which provides a globally usable reference to a unique object at each place. The issue of distributed garbage collection is solved using a *Global Object Tracker* which maintains a record of all remote references to globalized objects [Kawachiya et al., 2012].

X10 provides two compilation paths using source-to-source compilation: *Managed X10*, which generates Java code to run on a standard Java virtual machine (JVM) with an X10 runtime library; and *Native X10*, which compiles to C++ and is compiled to a

binary application using a standard C++ compiler. Communication between places is handled by *X10RT*, a C++ library implementation of the X10 runtime layer. There are several implementations of X10RT which use different communication methods including sockets, MPI, PAMI (for use on Blue Gene and Infiniband interconnects), and a stand-alone implementation to support multiple places on a single host. These implementations make it possible to run multi-place X10 programs on a wide range of computer systems.

While X10's parallel primitives support productive programming, they are not sufficient. Issues arise regarding efficient implementation of the primitives to provide acceptable performance on modern computer architectures; furthermore, productive programming requires reusable *distributed data structures* that support high-performance operations. These issues will be considered in chapter 3.

2.4.1.1 Active Messages in X10

X10 supports the *active message* idiom (see §2.1.3), which combines data transfer and remote computation within a single message; this allows for the terse expression of many distributed algorithms using localized patterns of communication and synchronization. This simple idea is at the core of the APGAS programming model: the X10 and Chapel languages have supported active messages since their inception, and active messages have been proposed as extensions for UPC [Shet et al., 2009], Co-Array Fortran [Scherer et al., 2010], and the Global Arrays library [Chavarría-Miranda et al., 2012].

In X10, the computation initiated by an active message may include synchronization with other activities running at the target place. The code in figure 2.7 uses active message synchronization to implement a single-slot buffer. Activity 1 sends an active message (lines 6–11) to the home place of a buffer, which fills the buffer with a value of type `T`. Activity 2, already running at the buffer's home place, waits for the buffer to be filled (line 16), and then removes and computes on the value.

In Chapter 3 we show how synchronization in active messages may be used to implement an efficient update algorithm for ghost regions in distributed arrays.

2.4.1.2 X10 Global Matrix Library

The X10 Global Matrix Library (GML) was developed to support distributed linear algebra in X10. It provides a variety of single-place and distributed matrix formats for dense and sparse matrices, and implements linear algebra operations for these matrix formats. High-level operations operate on entire matrices and are intended to support a programmer writing in a sequential style while fully exploiting available parallelism. Operations for single-place dense matrices are implemented as wrappers around the Basic Linear Algebra Subroutines (BLAS) [Blackford et al., 2002] and Linear Algebra PACKage (LAPACK) [Anderson et al., 1995] routines; more complex operations are implemented in X10 using the simple operations as building blocks. GML has been used to implement various machine learning algorithms including linear regression and PageRank [Page et al., 1999], and has been extended to tolerate loss of resources

```
1  val buffer:Buffer;
2  val bufferRef:GlobalRef[Buffer];
3  ...
4  // activity 1
5  val v:T = ...;
6  at(bufferRef.home) async {
7    val buffer = bufferRef();
8    when(!buffer.full) {
9      buffer.put(v);
10   }
11 }
12 ...
13 // activity 2 at buffer home
14 async {
15   val v:T;
16   when(buffer.full) {
17     v = buffer.remove();
18   }
19   computeOn(v);
20 }
```

Figure 2.7: X10 code using active messages to implement a single-slot buffer

at runtime [Hamouda et al., 2015]. GML is used in the work described in chapter 4 of this thesis.

2.4.2 Habanero Java

Early versions of the X10 specification were closely based on the Java programming language. This early design led to the development at Rice University of the Habanero Java language, an extension of Java for the APGAS programming model [Cavé et al., 2011]. Habanero Java includes the same parallel constructs as X10 (with some differences in naming), as well as sequential extensions for multidimensional arrays and efficient complex arithmetic. Where X10 provides clocks, Habanero Java provides a more general *phaser accumulator*, which is a unification of collective and point-to-point synchronization, including support for reduction [Shirako and Sarkar, 2010]. Yan et al. [2010] developed *hierarchical place trees* in Habanero Java, as a generalization of X10 places which maps more closely to the deep memory hierarchies on modern computer architectures. Habanero Java is an important vehicle for research into runtime implementation and compiler optimizations for APGAS languages; many techniques developed for it are directly applicable in X10, and vice versa.

2.4.3 Chapel

Chapel [Chapel, 2014] was developed by Cray Inc. to allow programmers to express a global view of parallel algorithms. Chamberlain et al. [2007] argue that this is a

necessary evolution from the fragmented view of computation inherent in the MPI programming model, in which programs are described on a per-task basis, explicitly dividing computation and data into per-task portions.

Chapel represents locality through the *locale* type. A locale represents one or more co-located processing elements with associated local storage, for example a single cluster node. Data and tasks can be specifically assigned to locales. Domain maps support arbitrary mappings from array indices to locales, allowing the decomposition of array data and computations. Figure 2.8 demonstrates the use of Chapel domain maps to distribute array data between locales. The code computes the DAXPY operation $\mathbf{Y} = \alpha\mathbf{X} + \mathbf{Y}$. Line 2 defines the domain for the distributed arrays x and y, block-distributed over locales. Line 5 uses scalar promotion of the multiplication and addition operators over the arrays to specify the operation in a single line of code. This is equivalent to a parallel loop over all indexes in the index set of the domain, performing the DAXPY operation on corresponding elements of the arrays. As in this simple example, Chapel is able to express many distributed array computations succinctly.

```
1  config const N = 1000000, alpha = 3.0;
2  const VecDom: domain(1) dmapped Block({0..#N}) = {0..#N};
3
4  proc daxpy(x:[VecDom] real, y:[VecDom] real):int {
5      y = alpha * x + y;
6  }
```

Figure 2.8: Chapel code demonstrating domain map and scalar promotion in computation of DAXPY over vectors

Chapel is designed on the principle of *multi-resolution*, meaning that high-level language features are implemented in terms of lower-level features that are not hidden from the programmer. For example, high-level data-parallel abstractions are implemented using the lower-level features of tasking and locality control. This permits the programmer to use high-level features for productive development, while retaining fine control for performance critical code [Chamberlain et al., 2011].

Chapel supports interoperability with other languages including C, C++, Fortran, Java and Python, through interface definitions for the Babel framework [Epperly et al., 2011]. Data structures such as local and distributed arrays may be shared between codes of different languages, but all communication is handled by the Chapel runtime [Prantl et al., 2011].

2.4.4 Fortress

Fortress [Allen et al., 2008] was developed by Sun under the HPCS program as a parallel programming language and framework for design of domain-specific languages. It compiles to Java bytecode and runs on an unmodified JVM, with the addition of Fortress runtime library code. The Fortress syntax is intended to mirror mathematical

notation, supporting Unicode characters and two-dimensional notation. Mathematical notation may be coded using ASCII shorthand notation and is typeset for display. For example, the definition of the factorial function [Allen et al., 2008]:

```
factorial(n) = PROD[i <- 1:n] i
```

would be rendered for display as:

$$factorial(n) = \prod_{i \leftarrow 1:n} i$$

This is intended to make it easier for scientists to compare code with formulae and switch between the two during development.

Fortress is implicitly parallel, intentionally requiring more effort to implement an explicitly sequential version of an algorithm. The primary form of parallelism is divide-and-conquer; for example, a **for** loop is recursively divided into sub-tasks that may be executed on different processing units. Explicit task parallelism is also supported. The Fortress runtime uses Cilk-style work stealing to balance load between processors.

Fortress provides information to programs about the execution environment through the *region* data structure. A *region* in Fortress holds information about hardware resources and relative communication costs. Aggregate data structures such as distributed arrays are mapped to regions through the mechanism of a *distribution*.

As the development of Fortress suffered from a lack of funding, and active development ceased in 2012, there is a lack of compelling application examples in scientific computing. Although a high-performance implementation of the entire language was never completed, Fortress included a number of interesting features, for example, its mathematical syntax and typesetting (jokingly described as 'run your whiteboard'), conditional inheritance and method definition, and integrating physical units into the type system. Some of these concepts may find future application in other languages for scientific and parallel programming.

2.5 Quantum Chemistry

Quantum chemistry uses quantum mechanical principles to model the nuclei and electrons that constitute a molecule. The time-independent Schrödinger equation,

$$\mathscr{H}\Psi = E\Psi, \tag{2.1}$$

is the fundamental equation of quantum chemistry and relates the energy of a system described by wavefunction Ψ to the Hamiltonian operator $\mathscr{H}$ over the constituent particles. The electronic Schrödinger equation describes the wavefunction of the electrons in a molecular system according to the Hamiltonian,

$$\mathscr{H}_e = -\frac{1}{2}\sum_i \nabla_i^2 - \sum_{i,A} \frac{Z_A}{r_{iA}} + \sum_{i>j} \frac{1}{r_{ij}}, \tag{2.2}$$

in which the first sum represents the kinetic energy of the electrons, the second represents Coulomb attraction between the electrons and nuclei, and the third sum represents repulsion between electrons. The electronic wavefunction of a molecular system allows the calculation of the potential energy surface for the nuclei and many important properties including stable conformations, active sites, ionization potentials, and spectral properties.

The Hartree–Fock approximation [Fock, 1930] expresses the electronic wavefunction $|\Psi\rangle$ in the form of a linear combination of permutations of one-electron *spin orbitals* χ_i, conveniently represented as a Slater determinant:

$$|\Psi\rangle = \frac{1}{\sqrt{N!}} \begin{vmatrix} \chi_1(1) & \chi_2(1) & \dots & \chi_a(1) & \dots & \chi_N(1) \\ \chi_1(2) & \chi_2(2) & \dots & \chi_a(2) & \dots & \chi_N(2) \\ \vdots & \vdots & & \vdots & & \vdots \\ \chi_1(N) & \chi_2(N) & \dots & \chi_a(N) & \dots & \chi_N(N) \end{vmatrix}. \tag{2.3}$$

Each spin orbital is the product of a spatial orbital ψ_i and a spin function α or β. The Hartree–Fock energy,

$$E_0 = \langle \Psi_0 \,|\, \mathscr{H} \,|\, \Psi_0 \rangle , \tag{2.4}$$

is an upper bound on the ground state energy of the system. According to the variational principle, the 'best' spin orbitals to use are those which minimize the Hartree–Fock energy [Szabo and Ostlund, 1989].

In this thesis, we will consider only restricted closed-shell Hartree–Fock calculations, in which each spatial orbital is doubly occupied by electrons of opposing spin values, hence the wavefunctions are calculated over spatial orbitals only. The spatial molecular orbitals ψ_i are usually expanded in the basis of a set of atomic orbitals ϕ_j,

$$\psi_i(\mathbf{r}) = \sum_j c_{ij} \phi_j(\mathbf{r}), \tag{2.5}$$

where the c_{ij} are the molecular orbital coefficients. Most quantum chemical calculations are performed using Gaussian-type atomic orbitals of the form:

$$\phi(\mathbf{r} - \mathbf{R}) = N(x - A_x)^k (y - A_y)^m (x - A_z)^n e^{-\alpha|\mathbf{r}-\mathbf{R}|^2}, \tag{2.6}$$

where $\mathbf{R}$ is the orbital center, N is a normalization factor and x, y, z are the Cartesian components of $\mathbf{r}$.

2.5.1 The Self-Consistent Field Method

The self-consistent field method (SCF) is a foundational method in quantum chemistry, and is widely used in computational chemistry packages such as GAMESS (US) [Schmidt et al., 1993], Gaussian [Frisch et al., 2009], NWChem [Valiev et al., 2010] and Q-Chem [Shao et al., 2013].

The SCF procedure requires the solution of the pseudo-eigenvalue problem [Szabo

and Ostlund, 1989]

$$FC = \epsilon SC, \tag{2.7}$$

where ϵ is a diagonal matrix of the orbital energies and F, C and S are the *Fock*, molecular orbital *Coefficient* and *Overlap* matrices respectively, the dimensions of which are dependent on the number of basis functions used to represent the molecular system, and therefore indirectly on the number of atoms in the system. The overlap matrix S is a constant Hermitian matrix composed of the overlap integrals between pairs of basis functions:

$$S_{\mu\nu} = \int d\mathbf{r}_1\, \phi_\mu^*(1)\phi_\nu(1). \tag{2.8}$$

The Fock matrix F is dependent on C, which necessitates use of an iterative procedure to solve (2.7). This procedure is defined by the so-called Roothaan–Hall equations [Roothaan, 1951; Hall, 1951]:

$$F_{\mu\nu} = H_{\mu\nu}^{\text{core}} + \sum_{\lambda\sigma} P_{\lambda\sigma}[(\mu\nu|\lambda\sigma) - \frac{1}{2}(\mu\sigma|\lambda\nu)] \tag{2.9}$$

$$P_{\mu\nu} = 2\sum_i C_{\mu i}C_{\nu i}^* \tag{2.10}$$

$$(\mu\nu|\lambda\sigma) = \int\int d\mathbf{r}_1\, d\mathbf{r}_2\, \phi_\mu^*(1)\phi_\nu(1)\frac{1}{r_{12}}\phi_\lambda^*(2)\phi_\sigma(2) \tag{2.11}$$

$$H_{\mu\nu}^{\text{core}} = \int d\mathbf{r}_1\, \phi_\mu^*(1)(\frac{1}{2}\nabla_1^2 - \sum_A \frac{Z_A}{r_{1A}})\phi_\nu(1) \tag{2.12}$$

Starting with an initial guess for the density matrix P, the procedure solves the density matrix that minimizes the energy for a given molecular system. The most expensive part in the SCF procedure is the creation of the Fock matrix (2.9), where μ, ν, λ, σ denote atom-centered basis function indices, $(\mu\nu|\lambda\sigma)$ is a four-centered two-electron integral (2.11), and H^{core} is a matrix containing one-electron integrals (2.12).

The Fock matrix represents the single-electron energy operator in the chosen basis set. It comprises two components: the Coulomb or J matrix representing the Coulomb operator,

$$\hat{J}_j(1) = \int \phi_j(2)^* \frac{1}{r_{12}} \phi_j(2)\, d\mathbf{r}_2\,, \tag{2.13}$$

and the Exchange or K matrix representing the exchange operator,

$$\hat{K}_j(1)\phi_i(1) = \left(\int \phi_j(2)^* \frac{1}{r_{12}} \phi_i(2)\, d\mathbf{r}_2\right) \phi_j(1). \tag{2.14}$$

Given these components and the one-electron Hamiltonian $\hat{h}(1)$, the Fock operator is defined [Szabo and Ostlund, 1989, §3.4.1] as

$$\hat{F}(1) = \hat{h}(1) + \sum_j \left[2\hat{J}_j(1) - \hat{K}_j(1)\right]. \tag{2.15}$$

In principle, formation of the Fock matrix is an $\mathcal{O}(N^4)$ operation, for a molecu-

lar system represented using N basis functions. For large molecules, however, the distances between centers means many integrals are small enough that they may be safely neglected (screened), reducing the actual cost of calculation to $\mathcal{O}(N^2)$ [Dyczmons, 1973]. For small molecules, the computed two-electron integrals may be stored and reused between iterations of the SCF, however, for larger molecules this is infeasible and the integrals must be directly evaluated on each iteration as they are built into the Fock matrix [Lüthi et al., 1991].

The individual two-electron integrals can be computed independently, but incur vastly different computational costs. Further, each two-electron integral contributes to up to six different locations in the Fock matrix [Lüthi et al., 1991]. Overall, this problem is characterized by load imbalance in the evaluation of the two-electron integrals and random scatters to memory elements in the F matrix. The SCF procedure also requires a matrix diagonalization of size N, and other linear algebra operations. For moderately sized systems, the cost of these operations is negligible compared to the cost of computing the F matrix.

While the fundamentals of the self-consistent field method may be expressed as familiar linear algebra operations, it presents two key challenges for parallel implementation for distributed systems:

- Key data such as the density matrix may be too large to fit in main memory of a single cluster node and must be distributed; however, these data must be globally accessible.
- Many intermediate quantities – the two-electron integrals – must be computed. The computation time required for each integral is highly irregular, requiring dynamic load balancing.

These challenges motivated the development of the Global Arrays library [Nieplocha et al., 2006a], which enabled one of the first distributed SCF implementations [Harrison et al., 1996].

Because of the fundamental nature of SCF to quantum chemistry and its inclusion in widely used software packages, it is a natural target application when considering programming models for scientific computing. Shet et al. [2008] explored the programmability of the Chapel, X10 and Fortress languages using SCF as an application example. They considered different methods for load balancing tasks for the creation of the Fock matrix including static, dynamic language-managed and dynamic program-managed load balancing. Their work focused on the expression of parallelism and array computations in the different languages. However, no concrete implementation of the same was provided due to the immaturity of the languages at the time of their work.

2.5.2 Resolution of the Coulomb Operator

The major factor limiting scaling of electronic structure calculations to larger problem sizes is the coupling between all pairs of electrons due to Coulomb and exchange interactions. Even after the application of screening techniques for large molecules,

there are $\mathcal{O}(N^2)$ two-electron integrals where N is the molecule size (number of atoms). Several linear-scaling methods have been proposed which have complexity $\mathcal{O}(N)$ [Ochsenfeld et al., 2007]. These include the Continuous Fast Multipole Method [White et al., 1996], which divides charge distributions hierarchically in a tree and approximates interactions between distant distributions by multipole expansions; and the KWIK algorithm [Dombroski et al., 1996], which partitions the Coulomb interaction into a short-range part to be solved directly, and a long-range part to be solved in Fourier space.

Limpanuparb [2012] showed that a two-center function like the Coulomb potential $L(r_{12}) = \frac{1}{r_{12}}$ or the long-range Ewald potential $L(r_{12}) = \frac{\mathrm{erf}(\omega r_{12})}{r_{12}}$ may be resolved into a sum of products of one-center spherical functions

$$L(r_{12}) = \sum_{n=0}^{\infty}\sum_{l=0}^{\infty}\sum_{m=-l}^{l} \phi_{nlm}(\mathbf{r}_1)\phi_{nlm}(\mathbf{r}_2) \equiv \sum_{k=1}^{\infty} \phi_k(\mathbf{r}_1)\phi_k(\mathbf{r}_2). \tag{2.16}$$

This infinite series may be truncated to a finite sum by truncating the radial resolution n after $\mathcal{N}'$ terms and the angular resolution after $\mathcal{L}$ terms:

$$L(r_{12}) \approx \sum_{n=0}^{\mathcal{N}'}\sum_{l=0}^{\mathcal{L}}\sum_{m=-l}^{l} \phi_{nlm}(\mathbf{r}_1)\phi_{nlm}(\mathbf{r}_2) \equiv \sum_{k=1}^{\mathcal{K}} \phi_k(\mathbf{r}_1)\phi_k(\mathbf{r}_2). \tag{2.17}$$

where $\mathcal{K} = (\mathcal{N}'+1)(\mathcal{L}+1)^2$ and $k = n(\mathcal{L}+1)^2 + l(l+1) + m + 1$. It is therefore possible to replace the calculation of two-electron integrals with the sum of products of one-electron overlap integrals:

$$(\mu\nu|\lambda\sigma) \approx \sum_{nlm}^{\mathcal{K}} (\mu\nu|nlm)(nlm|\lambda\sigma). \tag{2.18}$$

Limpanuparb et al. [2013] showed that both Coulomb and long-range Ewald potentials can be resolved into one-electron functions of the form

$$\phi_{nlm}(\mathbf{r}) = q_n j_l(\lambda_n r) Y_{lm}(\mathbf{r}), \tag{2.19}$$

where $j_l(t)$ is a spherical Bessel function and Y_{lm} is a real spherical harmonic, and provided efficient recurrence relations to compute the auxiliary integrals

$$(\mu\nu|nlm) = \int \chi_\mu(\mathbf{r})\chi_\nu(\mathbf{r})\phi_{nlm}(\mathbf{r})\, d\mathbf{r}, \tag{2.20}$$

and demonstrated that the accuracy of the resolution can be controlled using a single threshold parameter THRESH, which determines the number of terms $\mathcal{N}'$ and $\mathcal{L}$ at which to truncate the radial and angular parts of the resolution so as to guarantee an error in the energy of no more than $10^{-\mathrm{THRESH}}$. Substituting (2.18) into the definitions of J and K matrices (2.13)–(2.14) yields the resolution of the Coulomb operator (RO)

expressions:

$$J_{\mu\nu} \approx \sum_{nlm}^{\mathcal{K}} (\mu\nu|nlm) D^{nlm} \tag{2.21}$$

$$D^{nlm} = \sum_{\lambda\sigma}^{N^2} P_{\lambda\sigma}(nlm|\lambda\sigma) \tag{2.22}$$

$$K_{\mu\nu} \approx 2 \sum_{a}^{O} \sum_{nlm}^{\mathcal{K}} (\mu a|nlm)(nlm|a\nu) \tag{2.23}$$

$$(\mu a|nlm) = \sum_{\lambda}^{N} C_{\lambda a}(\mu\lambda|nlm) \tag{2.24}$$

Computation of the Fock matrix using RO requires evaluation of $\mathcal{O}(N^2\mathcal{K}O)$ integrals, where N is the number of basis functions and O is the number of occupied orbitals. For large molecules, the number of significant shell pairs scales linearly with the size of the molecule, and therefore screening methods may be used to reduce the computational complexity to $\mathcal{O}(N\mathcal{K}O)$ [Limpanuparb, 2012] — quadratic in the size of the molecule. When increasing the quality of the basis set for a given molecule, $\mathcal{K}O$ may be regarded as a constant; however, screening strategies are not as effective in reducing the number of significant shell pairs, meaning that computation time is still almost quadratic in the number of basis functions.

RO can be used to compute the full Hartree–Fock energy for a molecule, however, this is typically expensive in comparison to alternative methods. Instead, the energy may be partitioned into a short-range and a smooth long-range component, for example using the Ewald partition [Limpanuparb et al., 2013]. RO may be used to evaluate the long-range component, while the short-range component of the energy can be evaluated efficiently by other means such as screening methods [Izmaylov et al., 2006]. The RO method of computing contributions to the Fock matrix is the same whether computing long-range or full Hartree–Fock energy.

RO will be used in chapter 4 as the basis for a distributed implementation of a complete Hartree–Fock calculation.

2.6 Molecular Dynamics

Molecular dynamics considers the behavior of systems of atoms[2] interacting by means of classical forces. Simulating whole atoms rather than individual electrons and nuclei allows the treatment of larger numbers of particles and timescales than is possible with quantum chemistry.

A molecular dynamics simulation is governed by a force field that defines the different types of interactions between particles. Bonded interactions such as bond stretch, bond angle and torsional terms approximate the influence of valence electrons in chemical bonds. Non-bonded interactions such as van der Waals and Coulomb

[2] or *residues*, multi-atom portions of a larger molecule

forces approximate long-range effects between atoms that are not joined by chemical bonds. Molecular dynamics has successfully been used to treat a range of problems including molecular structure, dielectric constants of protein solutions, ion channel configurations, and enzyme free binding energies [Hansson et al., 2002].

2.6.1 Calculation of Electrostatic Interactions

Just as in quantum chemistry, the most expensive part of molecular dynamics simulation is typically the calculation of long-range Coulomb interactions (also known as electrostatic forces). The electrostatic potential of a particle with charge q_i at $\mathbf{r}_i$ due to n other charged particles is:

$$\Phi_i = \sum_{j=1}^{n} \frac{k q_i q_j}{|\mathbf{r}_j - \mathbf{r}_i|} \tag{2.25}$$

It is apparent from the above definition that a particle interacts with all other particles in the system, and furthermore that the strength of the interaction decreases only slowly with distance (as r^{-1} for the potential, and as r^{-2} for the force). Exact solution requires the calculation of forces and potentials between every pair of particles in the system, a complexity of $\Theta(N^2)$. Such a calculation quickly becomes infeasible for large numbers of particles. While purpose-built architectures can accelerate the pairwise calculations [Susukita et al., 2003; Shaw et al., 2009], the basic complexity remains unchanged. Some method of approximation is necessary to scale to larger systems; two such methods are the particle mesh Ewald method (PME) and the fast multipole method (FMM).

2.6.2 Particle Mesh Ewald Method

The Smooth particle mesh Ewald method (PME) [Darden et al., 1993; Essmann et al., 1995] is used to evaluate electrostatic interactions in systems with periodic boundary conditions. The electrostatic energy is partitioned as $E = E_{\text{dir}} + E_{\text{rec}} + E_{\text{corr}}$, where the direct energy

$$E_{\text{dir}} = \frac{1}{2} \sum_{n}^{*} \sum_{i,j=1}^{N} \frac{q_i q_j \,\text{erfc}(\beta |\mathbf{r}_j - \mathbf{r}_i + \mathbf{n}|)}{|\mathbf{r}_j - \mathbf{r}_i + \mathbf{n}|} \tag{2.26}$$

is the short-range interaction that is calculated directly, and the reciprocal energy

$$E_{\text{rec}} = \frac{1}{2\pi V} \sum_{\mathbf{m} \neq 0} \frac{\exp(-\pi^2 \mathbf{m}^2 / \beta^2)}{\mathbf{m}^2} S(\mathbf{m}) S(-\mathbf{m}) \tag{2.27}$$

is the long-range interaction that is calculated in reciprocal space. E_{corr} is a correction due to self-interactions.

The algorithm comprises several steps:

1. approximation of a *charge density field* Q by interpolating charges on a mesh of grid points using B-splines (see figure 2.9);

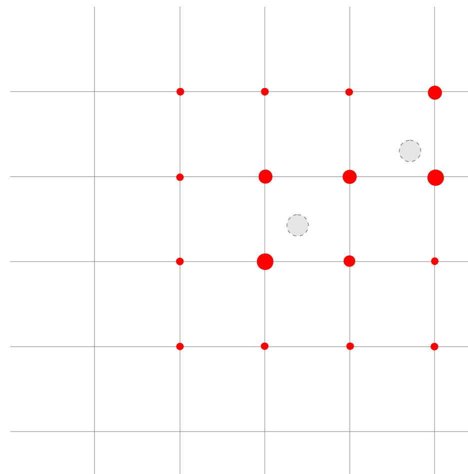


Figure 2.9: Charge interpolation in the particle mesh Ewald method: point charges (grey, dashed lines) are interpolated to nearby grid points (red).

2. calculation of the inverse FFT $F^{-1}(Q)$ of the charge array;
3. multiplication of the inverse charge array $F^{-1}(Q)$ with an array $F^{-1}(\Theta_{\text{rec}})$ representing the reciprocal space *pair potential*;
4. transformation of the result by a forward FFT to give the convolution of $\Theta_{\text{rec}} \star Q$;
5. calculation of the reciprocal potential as the entrywise product $(\Theta_{\text{rec}} \star Q) \circ Q$;
6. direct calculation of the short-range interaction within a reduced domain; and
7. correction to cancel the self-interaction of each particle.

Reciprocal forces and potentials are evaluated at grid points and interpolated to each particle. While steps 1–5 above must be performed in order, they are independent of steps 6 and 7 which may be performed in parallel. The core of the method is the two 3D fast Fourier transforms (FFTs) (inverse and forward) in steps 2 and 4 above. While the theoretical computational scaling is $\mathcal{O}(N \log N)$, a distributed 3D FFT requires all-to-all communication, which means that communication is the limiting factor in scaling the method.

Three factors affect the accuracy of the method:

- the cutoff distance for evaluation of direct interactions, where a longer cutoff increases the accuracy of the short-range component of the force/energy but requires more direct calculation and communication of more particle data;
- the order of B-spline interpolation. Typically each charge is interpolated over a small number of grid points (e.g. four) in each dimension, but higher-order interpolation increases the accuracy of the long-range component of the force/energy, at the cost of increased computation; and

- the grid spacing, where closer spacing increases the accuracy of the long-range component at the cost of increased computation and memory use. Typical grid spacings are on the scale of mean pair distances e.g. around 1.0 Å for molecular dynamics.

The choice of these parameters allows a tradeoff between speed and accuracy according to the requirements of the simulation.

There are a number of software packages for molecular dynamics (MD) that include implementation of PME. An efficient implementation is included in GROMACS [Pronk et al., 2013], which is a free software package for molecular simulation, primarily intended for use on small- to medium-sized compute clusters. GROMACS supports parallelism using both MPI and OpenMP. The Ewald partition is divided between two sets of nodes: many nodes evaluate the direct part while a smaller set evaluate the reciprocal part [Hess et al., 2013]. The evaluation of direct interaction (2.26) uses table-based interpolation of the complementary error function erfc, which gives higher performance than standard computation using series expansion. Reciprocal space evaluation including FFTs uses a 2D 'pencil' decomposition, and the number of nodes is limited to improve parallel scaling. GROMACS also supports other methods for the calculation of long-range electrostatics, including full pairwise interaction, cutoff-based interactions and Particle-Particle Particle-Mesh (P^3M).

2.6.3 Fast Multipole Method

The fast multipole method (FMM) [Greengard and Rokhlin, 1987] is of interest for large-scale molecular dynamics simulation, due to its low computational complexity of $\mathcal{O}(N)$ where N is the number of particles (compared to $\mathcal{O}(N \log N)$ for PME). FMM gives rigorous error bounds for the calculation of potential and forces, in contrast to particle-mesh methods, where only an error estimate may be available [Deserno and Holm, 1998]. This well-defined error behavior makes FMM a suitable choice for applications where provable accuracy is required. Also in contrast to particle-mesh methods, FMM does not require periodic boundary conditions, although it has been extended for such systems [Lambert et al., 1996; Kudin and Scuseria, 1998].

In the 3D FMM [Greengard and Rokhlin, 1997], the simulation space is divided into an octree: a tree of cubic boxes in which each box is recursively divided into eight child boxes. The recursion ends when either a maximum predefined tree depth $D_{\max}$ is reached, or the number of particles in a box is below a certain density threshold. Interactions between particles in nearby leaf boxes are evaluated directly, whereas distant interactions are evaluated by means of series expansions around box centers or equivalent densities.

There are many different choices of series expansion for the fast multipole method, including spherical harmonics [White and Head-Gordon, 1994], plane wave expansions [Greengard and Rokhlin, 1997], equivalent charges [Ying et al., 2004] and Cartesian Taylor expansions [Yokota, 2013]. The different expansions and operations have different computational and memory complexity with regard to the order of expansion (and consequent accuracy). Spherical harmonics with rotation-based translation

and transformation operations have the lowest asymptotic complexity, with relatively high sequential overhead [Yokota, 2013].

FMM is characterized by irregular, but highly localized data access patterns. The algorithm comprises several steps:

1. generation of multipole expansions from particles in each leaf box (*P2M*);
2. a post-order traversal[3] (upward pass) combining multipole expansions at higher levels in the tree (*M2M*);
3. transformation of the multipole expansions to local expansions for *well-separated* boxes (*M2L*);
4. a pre-order traversal[4] (downward pass) translating and adding parent local expansions to child boxes (*L2L*);
5. evaluation of far-field interactions for all particles in leaf boxes using the local expansion for each box (*L2P*); and
6. direct evaluation of near-field interactions between particles in non-well-separated boxes (*P2P*).

The fast multipole method allows *a priori* calculation of strict error bounds [White and Head-Gordon, 1994]. The tradeoff between accuracy and computation time is controlled by parameters to the algorithm:

- the number of terms p in the multipole and local expansions, where a larger number of terms results in greater accuracy and computational cost;
- the maximum depth of the tree $D_{\max}$ or the maximum number of particles per lowest-level box q, where a larger $D_{\max}$ (or smaller q) means more interactions are computed using far-field approximation rather than direct evaluation; and
- the definition of *well-separated* i.e. the number of box side lengths ws between a pair of boxes, where $ws = 1$ is the minimum and larger values of ws mean more interactions are computed using direct evaluation.

There are many published implementations of FMM, which differ in both algorithm and technology (programming language, libraries) used to implement them.

The exaFMM [Yokota, 2013] package is a state-of-the-art implementation of FMM that uses Cartesian Taylor expansions and is optimized for low accuracy (≤ 3 decimal digits) calculations. It is claimed as the fastest sequential implementation of the FMM [Taura et al., 2012], and as an order of magnitude faster than competing codes on a single Intel node [Yokota, 2013]. Single-precision arithmetic is used, and the number of terms, p, in expansions is a compile-time constant, which enables template meta-programming with specialization for low p. Although Cartesian Taylor

[3] In a post-order traversal, parent nodes in the tree are visited *after* their children.
[4] In a pre-order traversal, parent nodes in the tree are visited *before* their children.

expansions have a higher asymptotic complexity ($\mathcal{O}(p^6)$) compared to spherical harmonics ($\mathcal{O}(p^3)$), they are always faster for low p due to smaller prefactors. exaFMM has been parallelized using data-driven execution and work stealing within a single node [Ltaief and Yokota, 2012].

The Kernel-Independent Fast Multipole Method (KIFMM) is a generalization of the fast multipole method for any second-order elliptic partial differential equation [Ying et al., 2003]. Instead of analytic expansions of the kernel function, it requires only kernel evaluations, which are used to construct equivalent densities on a cube or sphere. A series of papers using KIFMM describe octree construction with the DENDRO multigrid library and show how FMM tree construction may be scaled to very large numbers of particles and processes [Sundar et al., 2007, 2008; Lashuk et al., 2009].

2.6.4 Molecular Dynamics Simulation of Mass Spectrometry

Mass spectrometry is used to identify charged chemical species within a sample of material by measuring differences between their motions within a magnetic or electric field. Within biological chemistry, Fourier transform ion cyclotron resonance mass spectrometry (FTICR-MS) is widely used as it allows high resolution between species, small sample sizes and a choice of ionization methods [Marshall et al., 1998]. The mass-to-charge (m/q) ratios of the species in a sample are determined by measuring the current induced by cyclotron motion of the ions in a radially confining magnetic field. The induced current signal is a superposition of sine waves corresponding to the cyclotron frequencies of each species; these frequencies are extracted by Fourier transform.

A charged particle moving at velocity v in a uniform magnetic field follows a circular path due to the Lorentz force. In theory the measured frequency for a species of ion corresponds exactly to the mass-to-charge ratio according to the cyclotron equation:

$$\nu_c = \frac{1}{2\Pi} \frac{q\mathbf{B}}{m}, \tag{2.28}$$

where q and m are the ion charge and mass and B is the strength of the confining magnetic field. In FTICR-MS, ions are constrained within a *Penning trap*, the key features of which are shown in figure 2.10. The trap combines a uniform magnetic field to confine the ions to radial motion with a non-uniform electric field that constrains axial motion.

In practice there are a number of features of the apparatus that can cause a reduction in the measured frequencies. It is infeasible to generate a radially symmetric electric field; angular non-uniformity in this field means the measured frequency is dependent on the cyclotron radius of the particles. Ions experience a drag due to interaction with the image charge on the detector plates. Furthermore, there is a space-charge effect due to repulsion between ions which is dependent on the confining fields, number of ions and the relative m/q of the different species. Computer

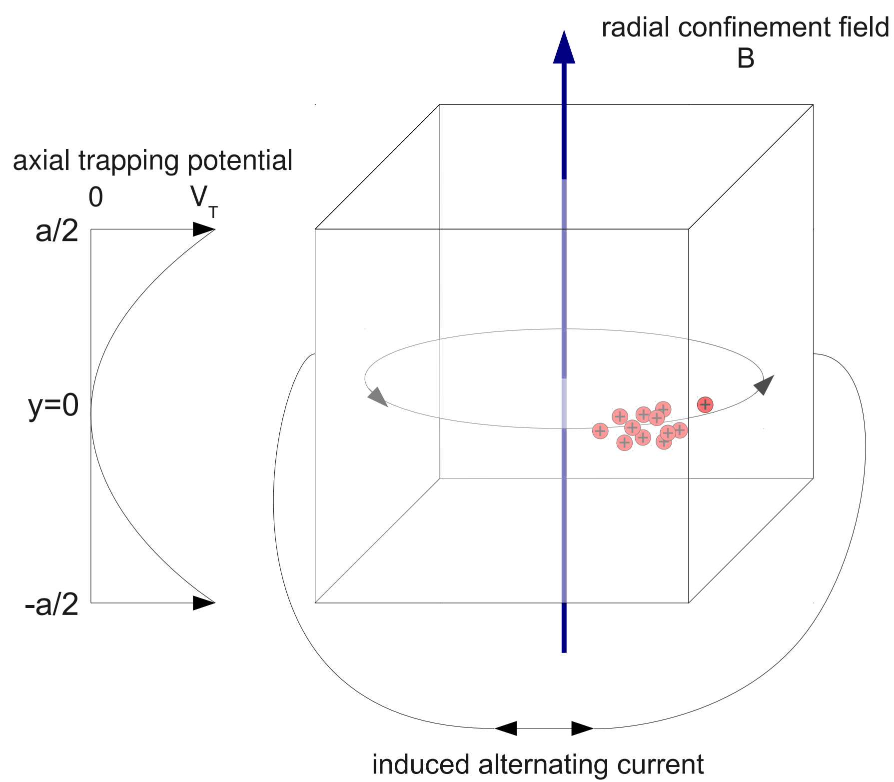


Figure 2.10: Fourier Transform Ion Cyclotron Resonance Mass Spectrometer: diagram of Penning Trap

simulation has aided in understanding these effects, as it allows precise control of system parameters that may be difficult or impossible to control in experiment.

In addition to the influence of the trapping field and the ion-image interaction, each ion experiences a repulsive Coulomb force from every other ion in the packet. As with molecular dynamics simulation discussed above, the calculation of these forces is nominally $\Theta(N^2)$, which makes the simulation of large ion packets infeasible unless some approximation is used to reduce the computational complexity.

Early simulations of ion trajectories in FTICR-MS focused on single-ion behavior, ignoring Coulomb interactions. Recent simulations model Coulomb interactions using either virtual particles [Fujiwara et al., 2010] or particle-in-cell approximations [Nikolaev et al., 2007; Leach et al., 2009; Vladimirov et al., 2011]. There appears to have been little consideration of the accuracy of electrostatic force evaluation. Chapter 5 will demonstrate the use of FMM for evaluation of ion-ion interactions, which allows the calculation of strict bounds on the truncation error for both force and potential calculations.

2.7 Application Patterns

Following an idea proposed by Colella [2004], researchers at the Berkeley Parallel Programming Laboratory identify a collection of thirteen 'dwarfs'[5] (see table 2.1) which represent patterns of computation and communication found in typical scientific and engineering applications. The dwarfs

> "... constitute classes where membership in a class is defined by similarity in computation and data movement. The dwarfs are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of applications. Programs that are members of a particular class can be implemented differently and the underlying numerical methods may change over time, but the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future." [Asanovic et al., 2006]

The co-design process described in this thesis uses application requirements to guide language and runtime design (and vice versa). It is therefore important that the applications used reflect real-world requirements for high-performance scientific computing. As a single co-design effort can only cover a small set of application codes, it is necessary to make generalizations from such a set to a broader range of applications.

[5] also known as 'motifs'

Table 2.1: The 'thirteen dwarfs': patterns of communication and communication in scientific and engineering applications. (Following Asanovic et al. [2006].)

Name	Description	Example Code
1. Dense Linear Algebra	Data stored as dense matrices or vectors. Unit stride memory access. Localized communication, row and column broadcasts.	ScaLAPACK [Blackford et al., 1996], Elemental [Poulson et al., 2013]
2. Sparse Linear Algebra	Data stored as compressed matrices with indexed loads and stores. Many integer operations for indexing.	PSBLAS [Filippone and Colajanni, 2000]
3. Spectral Methods	Data in frequency domain rather than spatial or time domain. Dependencies form butterfly patterns. All-to-all communications.	FFTW [Frigo and Johnson, 2005]
4. N-Body Methods	Interaction between pairs of points. Hierarchical methods group points to reduce complexity.	GROMACS [Hess et al., 2008], GADGET [Springel et al., 2001]
5. Structured Grids	Data represent a regular grid. High spatial locality. May be subdivided into finer grids (adaptive mesh refinement).	DENDRO [Sundar et al., 2007]
6. Unstructured Grids	Data represent an irregular grid with explicit connectivity. Multiple levels of indirection in data structure.	CHOMBO [Colella et al., 2012]
7. Map-Reduce	Large dataset; operations applied to independent elements and results aggregated.	[Hadoop]
8. Combinational Logic	Simple operations on large data; bit-level parallelism.	Hashing; encryption
9. Graph Traversal	Search over objects in graph structure; indirect memory access; little floating-point computation.	
10. Dynamic Programming	Compute a solution by solving simpler overlapping subproblems with optimal substructure.	Query optimization; DNA subsequence matching
11. Backtrack and Branch & Bound	Global optimization by pruning subregions.	Boolean satisfiability; combinatorial optimization
12. Graphical Models	Probabilistic models with random variables as nodes and conditional dependencies as edges.	Bayesian networks; neural networks
13. Finite State Machines	System of states connected by input-dependent transitions. Difficult to parallelize.	Compression; compilers

The dwarfs constitute a powerful classification framework which supports such generalizations. The applications discussed in this thesis correspond to three dwarfs, which are representative of important classes of scientific application: *dense linear algebra*, *spectral methods* and *N-body methods*. These dwarfs are described in detail in the following subsections. Recent application work in X10 is representative of other dwarfs including *sparse linear algebra* [Dayarathna et al., 2012a,b], *structured grids* [Milthorpe and Rendell, 2012], *MapReduce* [Shinnar et al., 2012]; *graph traversal* [Saraswat et al., 2011], and *dynamic programming* [Ji et al., 2012]. Taken as a whole, these efforts combined with the work described in this thesis are representative of a broad range of scientific and engineering applications.

2.7.1 Dense Linear Algebra

The most important and best understood class of computational patterns for scientific applications is dense linear algebra. These patterns use matrix data in a regular layout in memory, which is accessed using unit stride in one dimension and regular non-unit stride in the other. Standard APIs have been developed for commonly-used linear algebra operations, including the BLAS [Blackford et al., 2002] and LAPACK [Anderson et al., 1995] for which optimized implementations are available for all high performance architectures. Distributed linear algebra libraries such as ScaLAPACK [Blackford et al., 1996] and Elemental [Poulson et al., 2013] extend these efforts to distributed memory systems.

Quantum chemistry applications make heavy use of dense linear algebra, with tensor contractions, matrix multiplication, and inner products forming the key low-level operations of many algorithms. Chapter 4 will describe the use of dense linear algebra operations in the context of such an application, the self-consistent field method (SCF).

The ubiquity of dense linear algebra has led to its use in the High Performance Linpack [Petitet et al., 2008] as a metric for ranking high performance computing systems in the TOP500 rankings [TOP500]. High Performance Linpack performance is heavily dominated by floating-point computation. Alternative benchmarks have been proposed that stress other system characteristics such as communication subsystem [Graph500, 2010; Heroux and Dongarra, 2013]; however, given the ubiquity of dense linear algebra in scientific and engineering applications it is likely to remain the most important of the 'dwarfs' for the foreseeable future.

2.7.2 Spectral Methods

Spectral methods transform differential equations represented in spatial or time domains into a discrete equation represented in the frequency domain. Such transformations arise naturally in functions with natural periodicity such as signal processing applications. Within computational chemistry, spectral methods may be applied to problems that can be cast into periodic form, such as evaluation of long-range potentials and fields under periodic boundary conditions [Berendsen, 2007, s. 13.10]. The particle-particle particle-mesh (P^3M) method developed by Hockney and Eastwood

[1988] partitions the potential into a short-range component to be solved directly and a long-range component solved on a grid using spectral methods; variants of this method including PME discussed in §2.6.2 are the most commonly used methods in molecular dynamics simulation of liquids and gases.

Spectral methods are attractive for numerical simulation because their convergence is exponential in the number of series coefficients, minimizing the amount of memory required to achieve high accuracy approximations [Boyd, 2001, chapter 1]. As they operate on global representation of the data, however, they require all-to-all communication on distributed architectures, which may limit scaling.

2.7.3 N-body Methods

In molecular dynamics (MD), interactions occur between every pair of particles in the system. Such pairwise interaction defines molecular dynamics as an *N-body problem*, similar to the problem of gravitational interaction. Hierarchical methods such as particle-mesh and tree codes may reduce the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ or $\mathcal{O}(N)$. Application codes dealing with N-body problems have achieved impressive scaling results, including many winners of the Gordon Bell Prize for scalability and performance (for example [Hamada et al., 2009; Rahimian et al., 2010; Ishiyama et al., 2012]). MD therefore represents a flagship application for computational science, and an appropriate target for application co-design.

2.8 Summary

This chapter presented a broad overview of programming models for high performance computing, categorizing different approaches primarily by the memory model — the view of memory provided to the programmer. The partitioned global address space model was presented as a middle ground between the performance transparency of the distributed memory model and the ease of programming of the shared memory model. The chapter concluded with a discussion of the asynchronous PGAS model, which combines the flexibility of task-based parallelism with explicit recognition of data locality. The X10 programming language which is the focus of this thesis is representative of the asynchronous PGAS model.

This chapter also presented an overview of important classes of scientific and engineering application codes. Two examples were selected from the domain of computational chemistry: quantum chemistry and molecular dynamics simulation. The use of dense linear algebra, spectral and N-body methods make these codes representative of broader classes of scientific and engineering applications.

Chapter 3

Improvements to the X10 Language to Support Scientific Applications

This chapter examines features of the X10 programming language and the APGAS programming model that support the development of high performance scientific applications. Comparison is made with analogous features of alternative programming models including MPI and OpenMP.

In the course of this work, a number of improvements were proposed to the X10 language and runtime libraries, some of which have already been incorporated in public releases of X10, and some which are still under discussion. These improvements include:

- methods for managing and combining worker-local data and for visualizing locality of parallel tasks (§3.1);
- efficient implementation of active messages and extensions for collective active messages (§3.2); and
- efficient indexing of local data and support for ghost regions in distributed arrays (§3.3).

Portions of this chapter have been previously published in Milthorpe et al. [2011] and Milthorpe and Rendell [2012].

3.1 Task Parallelism

In a task-parallel programming model like APGAS, the program is divided into a set of tasks to be executed in parallel, each of which has its own execution environment and may follow a different execution path. In X10, these tasks are lightweight thread objects called *activities*. An activity may access data held at the current place, and may change place using the **at** statement to access data at another place. At any time a number of activities may be executing in parallel at a place.

Load balancing within a place is supported by a work stealing runtime [Tardieu et al., 2012]. The runtime maintains a pool of worker threads, each of which processes a double-ended queue (*deque*) of activities. A worker thread that creates an activity adds the activity to the head of its own deque, and once the worker completes an activity it takes the next activity to be processed from the head of its own deque. Thus activities are executed by each worker in a last-in, first-out (LIFO) order. When a worker becomes idle, it attempts to steal work from the tail of another worker's deque. The runtime uses cooperative scheduling in which an activity runs to completion unless it encounters a blocking statement or explicit yield [Grove et al., 2011].

As previously discussed in chapter 2, X10 explicitly represents locality in the form of places. However, issues of locality also arise in considering data access patterns within a place, in particular between worker threads executing on a single cache-coherent multicore node. Reuse of cached data between cores may increase performance, whereas false or unnecessary sharing of cached data may reduce performance. To make efficient use of the memory system of a place, the programmer must be able to understand the sharing of data within the cache hierarchy. The following subsections propose two extensions to X10: the first allows the programmer to allocate, manage and combine local copies of data; the second supports the visualization of tasks executed by each worker thread.

3.1.1 Worker-Local Data

In X10's APGAS model, any processing element may access memory anywhere in the global address space. Nevertheless, there are still cases in which it is desirable that multiple copies of data be maintained, including:

- place-local copies of remotely held data, to avoid communication costs of remote access; and
- worker-local copies of data, either to avoid synchronization costs (locking) between worker threads in a uniform memory area, or to avoid costs due to cache invalidation for worker threads operating on the same data.

In a distributed memory programming model such as MPI, processes may only access remote data by means of explicit messages. Therefore all data are effectively place-local, and thread-local data are an orthogonal concern.

Conversely, in a shared memory model such as OpenMP, there is no concept of remote access or associated communication costs. However, the importance of thread-local data has been recognized for the reasons described above. OpenMP clauses such as `private`, `firstprivate` and `lastprivate` specify that a thread-local copy of a variable should be created for each thread within a team, and allow the user to control how these are copied from or assigned back to shared data outside of a parallel region. The `reduction` clause allows an OpenMP implementation to create private copies of a reduction variable for each thread in a team, to avoid unnecessary synchronization within a parallel region.

For example, the following OpenMP code calculates a dot-product in parallel, which is reduced by a sum operation at the end of the parallel region. The implementation may create a private copy of the `result` variable for each thread.

```
1 double a[n], b[n], result;
2
3 #pragma omp parallel for reduction(+:result)
4 for (i=0; i < n; i++) {
5     result = result + (a[i] * b[i]);
6 }
```

In X10's APGAS model, both place-local and worker-local data are useful for different purposes. They are supported by the classes `x10.lang.PlaceLocalHandle` and `x10.util.WorkerLocalHandle` respectively.

A `PlaceLocalHandle` provides a unique ID that can be efficiently resolved to a unique local piece of storage at each place. Initialization of a `PlaceLocalHandle` is a distributed operation that runs a remote activity at each place to initialize the storage, and store a pointer to the storage under its unique ID in a lookup table (natively implemented as a C++ global variable, or a Java static field). `PlaceLocalHandle` is used to implement the `x10.regionarray.DistArray` class, by providing storage for the local portion of the array at each place. It may also be used directly in user code to construct distributed data structures, or to provide replicated data. For example, the following code replicates the array `coeffs` to each place, and then performs some computation at each place using the local copy:

```
1 val coeffs = new Rail[Double](N);
2 // ...
3 val coeffsHandle = PlaceLocalHandle.make(PlaceGroup.WORLD, () =>
    coeffs);
4 // ...
5 finish ateach(place in Dist.makeUnique()) {
6     val localCoeffs = coeffsHandle();
7     compute(localCoeffs);
8 }
```

Chapter 4 will describe how worker-local and place-local values can be used to divide a parallel quantum chemistry code into independent tasks for load balancing between processing elements. The existing `x10.lang.PlaceLocalHandle` class provides a suitable base for the implementation of distributed data structures such as distributed arrays, and was also found to be useful to the application programmer. In contrast, the original implementation of `WorkerLocalHandle` was found to be unsuitable and required modification as described in the following subsection.

3.1.1.1 Managing and Combining Worker-Local Data

Early versions of X10 included a class, `WorkerLocalHandle`, which provided worker-local storage for each worker thread at each place. However, it had a number of limitations which made it unsuitable for typical parallel patterns:

- it did not support initialization using a different value for each worker;
- there was no way to reset or reduce all worker-local data; and
- too many instances of worker-local data were allocated: instead of allocating one instance per active worker thread, `Runtime.MAX_THREADS` instances were created (the maximum number of worker threads that are permitted at the place, usually a large number e.g. 1000).

We proposed an improved version of `WorkerLocalHandle`, which provides a lazy-initialized worker-local store at each place. It also provides methods to apply a given closure to all local values at a place (which can be used to reset the storage) and perform a reduction operation over all local values. As instances are lazy-initialized, the cost of all-worker operations or reductions is kept to a minimum. Worker-local storage is limited to objects of reference type; this restriction is necessary to support lazy initialization. The basic structure of the improved `WorkerLocalHandle` is given in figure 3.1.

The X10 `Rail` class used for the `store` member variable (lines 5 and 8 of figure 3.1) is a zero-based, one-dimensional (C-style) array, which supports storage of a number of worker-local values up to the maximum number of worker threads that may be created by the runtime. The `initLocal` function (lines 24–29) sets the initializer function which is called whenever a worker-local value is lazy-initialized for a worker thread. When `initLocal` is called, it also clears worker-local values for all threads so that they can be reinitialized. The `reduceLocal` function (lines 46–55 of figure 3.1) can be used to perform a thread-safe reduction across worker-local values for all workers.

This improved version of `WorkerLocalHandle` makes it easy to allocate, reset and combine worker-local data. For example, the following code divides a region into a set of activities, one per element. Each worker thread computes a partial array result based on the activities that it handles (lines 2–5), and these are reduced using an array sum operation (lines 6–9).

```
1  val result_worker = new WorkerLocalHandle[Rail[Double]](
2      () => new Rail[Double](N)
3  );
4  finish for(i in region) async {
5      val partialResult = result_worker();
6      compute(partialResult, i);
7  }
8  val result = result_worker.reduceLocal(
9      (a:Rail[Double],b:Rail[Double]) => a.map(a, b,
10         (x:Double,y:Double)=>(x+y)) as Rail[Double]
11 );
```

The improved implementation of `WorkerLocalHandle` was incorporated into X10 version 2.4, and was used for all other performance measurements of X10 code presented in this thesis. Chapter 4 will describe how the improved `WorkerLocalHandle` including the `reduceLocal` function can be used to avoid synchronization in the calculation of auxiliary integrals in a quantum chemistry code.

```
1  public class WorkerLocalHandle[T]{T isref, T haszero}
2      implements ()=>T, (T)=>void {
3      private static class State[U]{U isref, U haszero} {
4          public def this(init:() => U) {
5              this.store = new Rail[U](Runtime.MAX_THREADS);
6              this.init = init;
7          }
8          public val store:Rail[U];
9          public var init:()=>U;
10     }
11     private val state:PlaceLocalHandle[State[T]];
12     public operator this():T {
13         val localState = state();
14         var t:T = localState.store(Runtime.workerId());
15         if (t == null) {
16             t = localState.init();
17             localState.store(Runtime.workerId()) = t;
18         }
19         return t;
20     }
21     /**
22      * Set init operation for this worker-local handle; clear current values
23      */
24     public def initLocal(init:()=>T):void {
25         val localState = state();
26         val localStore = localState.store;
27         localStore.clear(); // in case previously initialized
28         localState.init = init;
29     }
30     /**
31      * Apply the given operation in parallel to each worker-local value.
32      */
33     public def applyLocal(op:(t:T)=>void):void {
34         val localStore = state().store;
35         finish for (i in localStore) {
36             val t = localStore(i);
37             if (t != null) {
38                 async op(t);
39             }
40         }
41     }
42     /**
43      * Reduce partial results from each worker using the init operation to
44      * create the initial value and return combined result.
45      */
46     public def reduceLocal(op:(a:T,b:T)=>T):T {
47         val localState = state();
48         val localStore = localState.store;
49         var result:T = null;
50         for (i in 0..(localStore.size-1)) {
51             val t = localStore(i);
52             if (t != null) {
53                 if (result == null) result = t;
54                 else result = op(result, t);
55             }
56         }
```

Figure 3.1: An improved `x10.util.WorkerLocalHandle`: key features

3.1.1.2 New Variable Modifiers for Productive Programming

The direct use of the `PlaceLocalHandle` and `WorkerLocalHandle` classes as currently implemented is messy, as it requires the definition and initialization of each instance using 'boilerplate' template code. To support productive programming, we propose that the definition and initialization of `PlaceLocalHandle` and `WorkerLocalHandle` be supported by new variable modifiers for the X10 language: `placeLocal` and `workerLocal`. For example:

```
1 val t_worker:WorkerLocalHandle[T];
2 t_worker = new WorkerLocalHandle[T]( () => new T(params) );
```

could be written more simply as:

```
1 workerLocal t_worker:T;
2 t_worker = new T(params);
```

The addition of keywords to a language is not without cost: each keyword adds to the cognitive load for programmers learning and using the language. This cognitive load must be weighed against the alternative. In this case, the alternative is verbose boilerplate code that obscures the important information about the variable: its type and the operation used to initialize its local instances.

There is a symmetry in the relationship between `workerLocal` and **`async`** and `placeLocal` and **`at`**. Code within an **`async`** activity must assume that it is working a different instance of a `workerLocal` value, and furthermore it may assume that no other activity will read or modify that instance for the duration of the activity. Similarly, code within an **`at`** statement must assume that it is working on a different instance of a `placeLocal` value to its enclosing scope (and of course it may assume that code running at other places will not read or modify that instance). However, unlike `workerLocal` data, `placeLocal` data are not thread safe. Other worker threads at the place may access the `placeLocal` value, so possible concurrent accesses must be protected by **`atomic`** blocks.

The proposed new keywords are under discussion with the X10 core design team and have not been incorporated into the language.

3.1.2 Visualizing Task Locality In A Work Stealing Runtime

An optimal scheduler should place tasks on processing elements so as to maximize reuse of cached data. For X10 this means that as far as possible, activities that are 'nearby' in the sense of sharing data should be executed by the same worker thread. Divide-and-conquer style programs which share data between related subtasks tend to exhibit good locality when run on work-stealing runtimes. However, work stealing has been observed to exhibit poor locality for other applications, including iterative data-parallel loops, due to random stealing [Acar et al., 2000]. It may therefore be useful for the programmer to know the actual locality achieved by the work-stealing runtime during program execution. To support the profiling and visualization of task locality in X10, we propose a new annotation: `@ProfileLocality`.

Applying the `@ProfileLocality` annotation to an **async** statement causes the runtime to collect the following information for each activity that executes the statement:

- place ID;
- worker thread ID; and
- one or more locality variables specified by the programmer.

On termination of the program, the collected information for the full list of activities is printed in tabular format to the standard output stream. The `@ProfileLocality` annotation could be implemented by a simple compiler transformation, which would have the advantage that instrumentation and logging could be controlled using a compiler flag. For the purposes of evaluating the approach for this thesis, the transformation was performed by hand.

The following example code demonstrates the use of this facility to visualize the locality of activities in a one-dimensional domain decomposition. The code simply computes the histogram of an array of integers, by processing each element of the array in a separate activity.[1]

```
1  public static def compute(data:Rail[Int], numBins:Int) {
2      val bins = new Rail[Int](numBins);
3      for (i in 0..(data.size-1)) @ProfileLocality(i) async {
4         val b = data(i) % numBins;
5         atomic bins(b)++;
6      }
7      return bins;
8  }
```

The annotation on line 3 associates each activity generated by the loop with the locality variable `i`, which is simply the loop index. On termination, the program prints the locality variable `i` followed by the place ID and worker ID for each activity as follows:

```
# Histogram__closure__1
# i place worker
  0     0      1
  1     0      3
  2     0      2
  3     0      4
  4     0      2
  5     0      2
  6     0      2
...
```

The above output shows that the array element with index $i = 0$ was processed at place 0 by worker thread 1; array element $i = 1$ was processed at place 0 by worker

[1] This code is included in the public X10 distribution as the *Histogram* sample code.

Figure 3.2: Locality of activity-worker mapping for *Histogram* benchmark (n=2000, X10_NTHREADS=4).

Figure 3.3: Locality of activity-worker mapping for *Histogram* benchmark with divide-and-conquer loop transformation (n=2000, X10_NTHREADS=4).

thread 3, and so on. These data can then be visualized using standard plotting tools. For example, figure 3.2 shows the locality of activities generated by executing *Histogram* for an array of 2000 elements on four worker threads at a single place. The activities executed by each thread are represented by a unique color.

It is apparent from the diagram that both load balancing and locality are poor: most activities are executed by one particular worker thread, and the remaining activities are finely and randomly divided between the other three worker threads.

To see the value of such a visualization, contrast figure 3.2 with figure 3.3. In X10 version 2.3, the X10 compiler was enhanced to include an optional loop transformation which converts a **`for`** loop over an integer index to a divide-and-conquer pattern of activities, using recursive bisection. This significantly improves locality when using work stealing with simple **`for`** loops. Figure 3.3 shows the locality of activities generated by *Histogram* compiled using the divide-and-conquer loop transformation. The activities are now more evenly divided between worker threads, and each worker executes large contiguous chunks of the loop, meaning that its reads from the `data` array have a unit stride, rather than being scattered across memory.

Chapters 4 and 5 will show how visualizing the locality of activity execution can be used to inform parallel algorithm design for more complex scientific codes.

3.2 Active Messages

An active message combines data transfer and computation in a single message [von Eicken et al., 1992]. Active messages are the core mechanism for point-to-point communications in X10. The following sections describe performance improvements for the implementation of active messages in X10, and a generalization of active messages to collective communications between activities executing at different places.

In later chapters, these improvements will be demonstrated in the context of two scientific applications. Firstly, chapter 4 will describe the use of **`finish/ateach`** collective active messages to share work between places in a quantum chemistry application. Secondly, chapter 5 will demonstrate the use of `TreeCollectingFinish` to perform an energy summation over a group of places in a molecular dynamics application. Both applications make use of improvements to the serialization of active

messages described in the next section.

3.2.1 Serialization of Active Messages

As active messages are the primary communication mechanism in X10, their efficient implementation is critical to performance. Certain messages, for example basic reads or writes, may be simple enough that they can be mapped directly to hardware-supported RDMA operations [Tardieu et al., 2014]. However, in general an active message is implemented as a message to the target place, containing a closure identifier and the environment for the closure. The X10 compiler generates a unique closure for the body of each active message in the application. Each variable that is captured in the closure environment for an active message is deep copied [Saraswat et al., 2014, §13]. The variable is treated as the root of an object graph which is serialized into a byte stream to be included in the message, and is deserialized at the target place to create a copy of the object graph.

By default, the X10 compiler generates serialization and deserialization code for every class in the application. Serialization proceeds recursively, with the `serialize()` method for a class calling the `serialize()` method for each of its fields. The programmer can specify that a field is not to be included in the copy by specifying the field as **`transient`**. The programmer may also override the standard serialization mechanism for a class by implementing the `CustomSerialization` interface:

```
1 package x10.io;
2 public interface CustomSerialization {
3     /** @return the value that should be serialized */
4     def serialize():SerialData;
5 }
```

By the mechanisms described above, X10's serialization framework provides a useful default, as well as the ability to override the default for performance fine-tuning. Due to its general-purpose design, the original X10 serialization framework entailed overheads which were unnecessary for many common applications and architectures, specifically, in byte-order swapping and preserving identity relationships in copied object graphs. In §3.2.1.1 we describe the elimination of byte-order swapping to improve the overall performance of serialization (both default and programmer-specified) for typical architectures, and in §3.2.1.2 we propose a new annotation by which a programmer may fine-tune serialization to avoid the cost of maintaining object graphs.

3.2.1.1 Byte-Order Swapping

To ensure message compatibility between places of different architectures, X10 implements byte-order swapping. The standard message format uses big-endian byte order. When a place running on a little-endian architecture (e.g. x86) serializes data to be sent to another place, it swaps the byte order to big-endian during serialization. Similarly, when a little-endian place receives a message, it swaps the byte order to little-endian during deserialization.

```
1  public class BenchmarkSerializeRail {
2      static N = 1000000;
3      static ITERS = 1000;
4
5      public static def main(args: Rail[String]): void = {
6          val a = new Rail[Double](N);
7          val start = System.nanoTime();
8          for (i in 1..ITERS) {
9              at(here.next()) { a(0) = 1; }
10         }
11         val stop = System.nanoTime();
12         Console.OUT.printf("send rail: %10.3g ms\n",
13             ((stop-start) as Double) / 1e6 / ITERS);
14     }
15 }
```

Figure 3.4: X10 benchmark code for serialization: send `Rail` of 1M elements

This byte swapping was found to be a major contributing factor to the cost of communications. As a demonstration, the benchmark code in figure 3.4 measures the average time to send a `Rail` of 1 million doubles to a neighboring place. When executed on two single-threaded places, both running on the same Sandy Bridge Core i7-2600 CPU, it takes an average of 8.78 ms to send the rail, of which almost 4 ms is spent in byte swapping.

Byte swapping adds no value for systems where all places use the same byte order. To avoid this cost, we modified the X10 runtime to avoid byte swapping for homogeneous clusters. If the flag `-HOMOGENEOUS` is set during compilation of the X10 runtime, the byte-order swapping code is replaced with a straightforward memory copy. This reduces the time for the benchmark in figure 3.4 to from 8.78 ms to 4.84 ms. The simple `-HOMOGENEOUS` build modification for the X10 runtime was incorporated as the default for X10 version 2.3, and was used for all other performance measurements of X10 code presented in this thesis.

A more general solution would perform byte swapping between places only when necessary. For example, places could broadcast architectural information to all other places, and then use this information to determine whether to swap byte order in serialization. Alternatively, each message could include a flag (little/big-endian) which could be used at the receiving place during deserialization. On deserialization, byte order is swapped only if the message flags is different to the current architecture. This would slightly increase message size, but would avoid the need for places to know the architecture of other places and so may better support dynamic reconfiguration and scaling to very large numbers of places.

3.2.1.2 Object Graphs and Identity

Serialization aims to recreate a copy of the object graph at the receiving place with the same identity relationships between objects. Preserving the identity relationships is critical if the object graph contains cycles, or multiple pointers to the same object. During serialization, the local address of each object is converted to a unique ID that is used within the message in place of a pointer. Whenever the serializer encounters an object address, it looks up the address in a map to determine whether the object has already been serialized, and if so, the previously used ID is reused.

For many scientific applications, the data structures to be transferred are much simpler: the object graph is a simple tree. It is wasteful to perform address mapping for these structures; each reference may be treated as a unique instance and serialized in full. To improve performance for simple tree structures, the `@Tree` annotation is proposed. Any local variable or class field annotated `@Tree` would be flat-copied for all **`at`** statements, treating each object address as unique within the subgraph of which that field or variable is the root. Correct operation of this mechanism assumes:

- the object subgraph is acyclic; and
- each object is referenced only once.

As with other safety checks in X10[2], `@Tree` could be compiled in two modes. If an object graph were serialized at runtime with checking enabled, object addresses would be written to an address map as normal, and if a previously serialized address were found within an `@Tree` subgraph, an exception would be thrown. Object identifiers for the `@Tree` subgraph would not be included in the messages and therefore would not be used in deserialization. If an object graph were serialized with checking disabled, addresses within a `@Tree` subgraph would simply be ignored. This would leave open the possibility of duplicate serialization of an object, or an infinite loop in serialization; it would be the programmer's responsibility to ensure the contract of `@Tree` is fulfilled, to avoid such possibilities.

The `@Tree` proposal is under discussion with the X10 core design team and has not been incorporated into a public release of X10.

3.2.2 Collective Active Messages Using `finish/ateach`

In MPI, data movement collective operations provide a means for communicating related data between all processes in a group. Collective operations may take advantage of hardware support for efficient communications, for example: multicast operations on Infiniband networks [Hoefler et al., 2007]; the global collective network on Blue Gene/L and Blue Gene/P [Faraj et al., 2009]; and message unit collective logic and barrier control registers on Blue Gene/Q [Chen et al., 2011]. In this section, we propose that active messages should be extended to operate within groups of processes, in an analogous way to MPI collective operations. To demonstrate the

[2] For example: array bounds checks and place checks (checks to ensure that references to distributed data may only be dereferenced at the home place).

concept, broadcast and reduction active messages are implemented within the X10 compiler and runtime and evaluated using microbenchmarks.

Since its inception, X10 has included the **ateach** construct to support parallel iteration over a distributed array. To date this has been considered to be syntactic sugar — merely a convenient shorthand way of representing parallel iteration, which is translated by the compiler into a more verbose pattern of distributed activities [Saraswat et al., 2014, §14.5]. This transformation is performed automatically by the compiler as shown in figure 3.5; the code on lines 2–4 of figure 3.5(a) is transformed into the code in figure 3.5(b).

```
1 val D:Dist;
2 ateach(place in D.places()) {
3     S(p);
4 }
```

(a) original code

```
1 for (place in D.places()) at(place) async {
2     for (p in D|here) async {
3         S(p);
4     }
5 }
```

(b) compiler-transformed code

Figure 3.5: X10 version 2.4 compiler transformation of **ateach**

The code in figure 3.5(b) creates a number of distributed asynchronous activities which are not guaranteed to terminate before exit of the loop nest. To wait for termination, the **ateach** construct is typically used within an enclosing **finish**. For example, `finish ateach(place in group)S` executes the statement `S` at every place in a group of places. The execution of the above code requires the following steps:

1. an active message for the closure `S` is sent to each place in the group;
2. each place `P` executes `S`;
3. once execution of `S` is complete at place `P`, `P` sends a finish notification to the root place; and
4. the root place collects the finish notification and confirms that all activities in the scope of the finish have terminated.

Once the root place detects termination for all activities within the **finish**, execution of its enclosing activity continues.

We observe that the semantics of **ateach** do not require that an active message from the root be sent *directly* to each other place in the group, nor is it necessary that

```
1  static def runOrForwardAteach(level:Int, group:PlaceGroup, treeDepth
       :Int, body:()=>void):void {
2      if (level < treeDepth) {
3          // keep forwarding until tree is spanned
4          val groupSize = pg.size();
5          // right child is here.id + 2**level
6          val rightChildIndex = leftTreeIndex + Math.pow2(level);
7          if (rightChildIndex < groupSize) {
8              val rightChild = group(rightChildIndex);
9              val closure = ()=> {
10                 runOrForwardAteach(level+1, pg, treeDepth, body);
11             };
12             x10rtSendMessage(rightChild.id, closure, prof);
13         }
14         // left child is here
15         runOrForwardAteach(level+1, pg, treeDepth, body);
16     } else {
17         // then execute here
18         execute(new Activity(body, state));
19     }
20 }
```

Figure 3.6: Implementation of tree-based `ateach` in x10.lang.Runtime

the finish notification return directly to the root. This observation allows the use of a tree pattern of communication to implement both `ateach` and `finish`.

3.2.2.1 A Tree-Based Implementation of `finish/ateach`

A tree-based pattern of active messages was used to implement `ateach` in the class `x10.lang.Runtime`. Figure 3.6 shows a simplified version of the implementation. The `PlaceGroup` is mapped to a binary tree of treedepth $= \lceil \log_2(\text{size}(\text{group})) \rceil$ levels, where each place is a leaf node. Each node forwards the active message downwards through the tree (lines 3–15) until it reaches the leaf nodes. The body of the message is then executed at each place (line 18).

The same pattern of communications was used in reverse to implement a new class, `x10.lang.Finish.TreeFinish`, which handles distributed termination detection of activities within a binary tree of places. A subclass, `TreeCollectingFinish`, supports the collection and reduction of the results of all activities. In combination, tree-based `ateach` and `finish` can be used to effect a broadcast-reduction pattern of computation. However, as they are implemented using active messages they may be integrated with other activities in progress at each place and are therefore more flexible than MPI broadcast and reduction.

To evaluate tree-based `finish/ateach`, its performance is measured against that of the default implementation of `ateach` on two compute clusters: *Vayu* and *Watson 4P* (both described in appendix A). Figure 3.7 shows the benchmark code used to

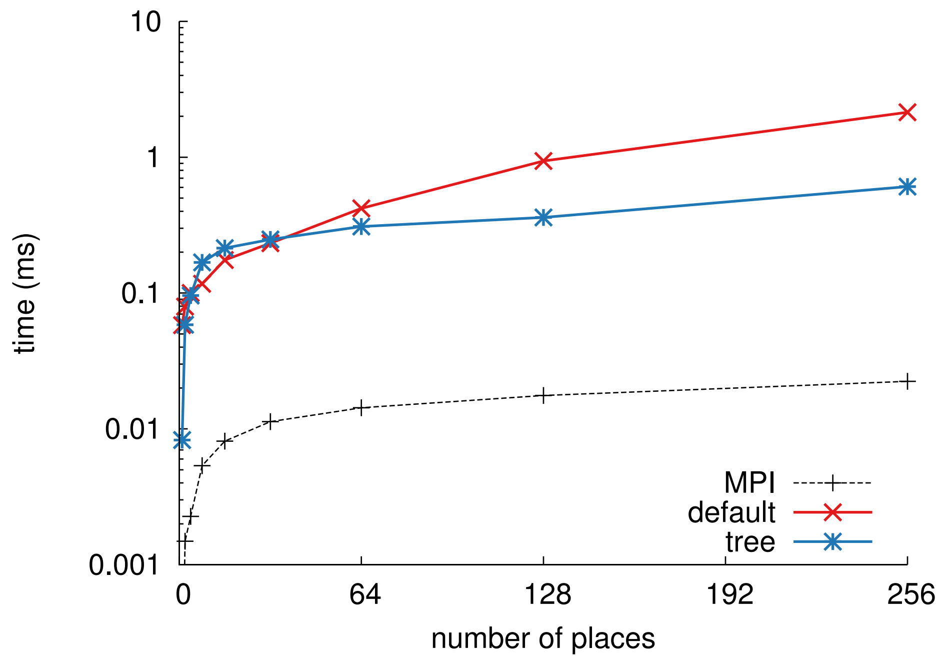
```
1  val a = DistArray.make[Short](Dist.makeUnique());
2  val dummy = 1S;
3  finish ateach([p] in a) { Team.WORLD.barrier(here.id); }
4  val start = System.nanoTime();
5  for (i in 1..ITERS) {
6      finish ateach([p] in a) {
7          a(p) = dummy;
8      }
9  }
10 val stop = System.nanoTime();
```

Figure 3.7: X10 benchmark code for **finish/ateach**

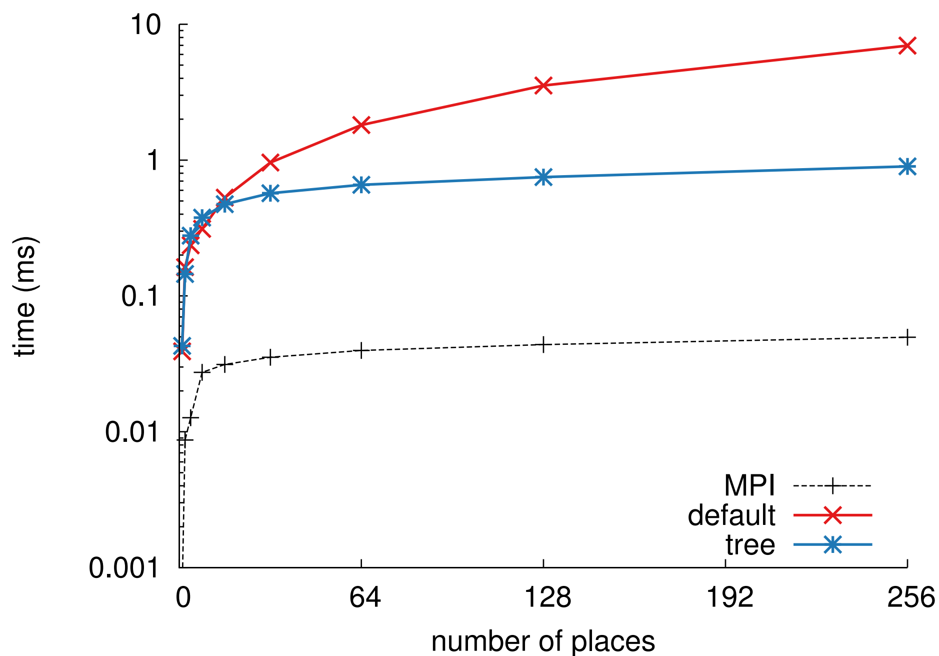
measure both implementations of **finish/ateach**. As a comparison, the scaling of MPI_Bcast/MPI_Reduce is measured for messages of one byte. The basic MPI broadcast benchmark was copied from a benchmark suite distributed by Intel [Intel, 2013b], however as the benchmark was modified, results from it should **not** be considered to be results from that suite. The MPI_Bcast benchmark was modified to add a call to MPI_Reduce directly after the broadcast call in each iteration. One X10 place or one MPI process was run per socket of *Vayu*/ core of *Watson 4P*.

Although MPI_Bcast is not an active message and does not initiate computation, it does transfer data from a root process to all processes in the group and so could be considered the 'ideal' against which performance of **ateach** should be compared. Similarly, MPI_Reduce collects data from a group of activities at a single root place, and so it is the ideal against which to compare the pattern of termination messages in a tree-based **finish**.

Figure 3.8 shows scaling of **finish/ateach** with the number of places on *Vayu* and *Watson 4P*. The absolute performance is about twice as fast on *Vayu* compared with *Watson 4P*; this is because serialization/deserialization of active messages is faster on *Vayu*'s more powerful Intel cores than it is on Blue Gene/P. The observed scaling of **finish/ateach** with the default implementation is $\mathcal{O}(N)$ on both platforms, whereas the tree-based implementation scales as $\mathcal{O}(\log N)$. The tree-based implementation is better for more than 64 places. While the improvement from using tree communication for **finish/ateach** is substantial, the performance of MPI_Bcast-MPI_Reduce is much better than either implementation of **finish/ateach** — around twenty times faster for large numbers of places. This suggests that further optimization of the hand-written tree implementation is required to match the performance of MPI_Bcast. However, the improvement in scaling from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$ makes it feasible to use this implementation of **ateach** to execute relatively long-running activities at each place within a group.



(a) *Vayu*: one place/process per socket



(b) *Watson 4P*: one place/process per core

Figure 3.8: Scaling with number of places of the **`ateach`** construct on *Vayu* and *Watson 4P*, and comparison with MPI broadcast.

3.3 Distributed Arrays

A distributed array stores portions of the array data at each place, while allowing global access to any element. Language support for distributed arrays is found in almost all PGAS languages, with the exception of Titanium in which all distributed data structures must be constructed using global pointers [Yelick et al., 2007b]. In X10, the `DistArray` class maps each point in a `Region` to a place in the system [Charles et al., 2005]. Each place holds a local portion of the array comprising those elements that are mapped to that place. X10 distributed arrays are very general in scope: they allow for arbitrary distributions (for example: block; block-cyclic; recursive bisection; fractal curve) and arbitrary regions (for example: dense or sparse; rectangular, polyhedral or irregular).

The flexibility and expressiveness of X10 distributed arrays makes them attractive from the point of view of productivity. For example, X10 allows programmers to develop algorithms that apply to arrays of arbitrary dimension while avoiding complex and error-prone indexing calculations [Joyner et al., 2008]; modifying the domain decomposition for a given array can be as simple as dropping in a different distribution class. However, there are significant challenges in implementing distributed arrays to achieve acceptable performance. In the following sections, we consider efficient methods for indexing distributed array data, and demonstrate how these methods can be used to build a high-level algorithm for updating ghost regions in distributed grid applications.

3.3.1 Indexing of Local Data

The generality of `DistArray` comes at a cost; it is implemented using standard object-oriented techniques and therefore requires key operations on the array to be implemented as virtual method calls. For example, the `Dist.offset(Point)` method determines the offset in memory for a particular element, from the start of the local storage for the current place. This is a virtual method as the calculation of the offset will be different depending on the distribution; however, many applications require only dense rectangular regions, and the cost of a virtual function call on every element access would be prohibitive. For this reason, X10 version 2.4 added a set of 'basic array' classes in a flat class hierarchy in the package `x10.array`. The basic array classes are final classes specialized to particular dimensions and distributions (e.g. `DistArray BlockBlock2` for a two-dimensional, dense, rectangular, zero-based, 2D-block-distributed array). This design avoids virtual function calls for array operations and minimizes space overheads [Grove et al., 2014].

Despite the obvious utility of the `x10.array` classes, some applications will still require the full generality of `x10.regionarray.DistArray`. To support these applications, we enhanced `DistArray` by adding a new method `getLocalPortion()`. This returns the local portion at the current place as an `Array`; if the local portion is not rectangular, it throws an exception. As `Array` is a final class, the local portion can be operated on without the overhead of virtual method calls.

In a similar vein, it is sometimes necessary to transfer a single message composed of multiple non-contiguous elements of a distributed array. For example, in a distributed linear algebra operation, a place may need to get a sub-block of a matrix from another place. The sub-block can be transferred as a standard X10 array, however, to fill the array it is necessary in general to copy the individual elements from the `DistArray`. Again in this case it is undesirable to incur the overhead of a virtual method call for each element access. To support efficient access to subregions of an array, we implemented a new method `DistArray.getPatch()`. This method efficiently copies a given region of the `DistArray` into a new `Array` object, inlining the offset calculation so as to avoid calling the virtual `offset(Point)` for each element.

The approach outlined above supports efficient indexing of local data for rectangular, block-distributed arrays, which are the most commonly used style of distributed array for scientific applications. However, a more general approach would be required for efficient indexing of arrays of arbitrary regions and distributions.

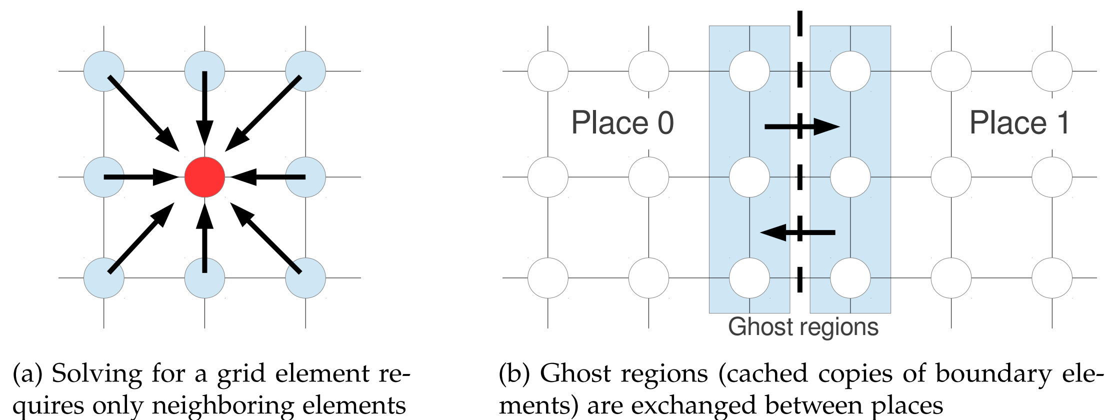
The `getLocalPortion()` method was incorporated into the array library in X10 version 2.2.1. Chapter 5 shows how `DistArray.getLocalPortion()` can be used in a particle mesh code for efficient iteration over the local portion of the mesh at each place.

`DistArray.getPatch()` was incorporated into the array library in X10 version 2.5.1. Our initial implementation requires hard-coding a separate version of the `getPatch()` method for each targeted number of array dimensions (1, 2, 3, ...), however it would be preferable to use automatic inlining and scalar replacement transformations [Fink et al., 2000] to generate efficient code for any dimension from generic X10 code [Milthorpe and Rendell, 2012].

3.3.2 Ghost Region Updates

Many physical problems can be modeled in terms of a system of partial differential equations over some domain. Discretizing the domain using a grid, and solving these equations for each grid point, involves computation over a large number of elements. Computation on each grid element requires only those elements in the immediate neighborhood as shown in figure 3.9(a); this locality property may be exploited in distributed algorithms.

A common feature of many distributed grid codes is the use of ghost regions. A ghost region holds read-only copies of remotely-held data, which are cached to enable local computation on boundary elements (as shown in figure 3.9(b)). Grid applications are typically iterative, so processes coordinate to exchange and cache ghost data many times over the course of a computation. The efficient implementation of ghost region updates is therefore an important factor in achieving good performance and scalability. For productive application development, it is important that ghost region updates be implemented using the standard distributed data structures available to the programmer, rather than being custom-built for each new application. Chapter 5 will show how an efficient ghost update algorithm can be used for productive programming and high performance in a distributed particle mesh application.



(a) Solving for a grid element requires only neighboring elements

(b) Ghost regions (cached copies of boundary elements) are exchanged between places

Figure 3.9: Solution of a system of partial differential equations on a grid

3.3.2.1 Implementing Ghost Region Updates for X10 Distributed Arrays

Support for ghost regions was implemented in the package `x10.regionarray` as a number of new classes, as well as modifications to the existing `DistArray` class. A new method on the `Region` class, `getHalo(haloWidth:Int)`, returns a halo region comprising the neighborhood of the target region. For rectangular regions, the halo region is simply a larger rectangular region enclosing the target region. For the special case of a zero-width ghost region (no ghosts), `Region.getHalo(0)` returns the region itself.

The constructor for `DistArray` was changed to allocate storage for the ghost region in `LocalState`. A new field, `LocalState.ghostManager:GhostManager`, holds a distribution-specific object that manages ghost updates. This reduces to standard behavior for `DistArray` in the special case of `ghostWidth==0` as there is no ghost manager and the ghost region is identical to the resident region. All operations on `DistArray` were changed to use the ghost region rather than the resident region for indexing.

The implementation of the `GhostManager` interface is specific to the distribution type. It may also use different algorithms depending on the target architecture. The initial implementation is for rectangular, block-distributed arrays only. For these arrays, ghost region data are collected for sending to each place using the `getPatch` method described in §3.3.1.

The following methods are defined on `DistArray` and constitute the user API for the ghost region implementation:

- `sendGhostsLocal()`

  an operation called at each place in the distribution that sends boundary data from this place to the ghost regions stored at neighboring places

- `waitForGhostsLocal()`

an operation called at each place in the distribution that waits for ghost data at this place to be received from all neighboring places

- `updateGhosts()`

  an operation that is called at a single place to update ghost regions for the entire array; this starts an activity at each place in the distribution to send and wait for ghosts

Low-Synchronization Algorithm for Ghost Updates

Ghost region updates are typically used in the context of a phased computation, for example:

```
1 for (i in 1..ITERS) {
2   updateGhosts();
3   computeOnLocalAndGhostData();
4 }
```

It is necessary to synchronize between neighboring places in a computation to ensure that all ghost regions have been fully received at a place before computation begins at that place.

There are two basic approaches to this problem. One is to use two-sided (send/receive or scatter/gather) communications. This is the approach used in the PETSc library [Balay et al., 2011], and in the M_P (message-passing) algorithm described by Palmer and Nieplocha [2002].

An alternative is to use one-sided communications surrounded by explicit synchronization. In some computations, such synchronization may naturally be included in the computation, for example to calculate a minimum or maximum value across all grid points. In the following example, collective synchronization occurs each iteration before the ghost data are sent and again before they are used in computation:

```
1 for (i in 1..ITERS) {
2   // collective synchronization
3   sendGhosts();
4   computeOnLocalData();
5   // collective synchronization
6   computeOnGhostData();
7 }
```

The collective operations surrounding the ghost update ensure the consistency of ghost data by enforcing an ordering with regard to other messages. All previous send operations from a place must complete before the collective reduction can begin. Where such natural synchronization is not present, the ghost update operation must perform synchronization before and after sending ghosts. In Global Arrays [Nieplocha et al., 2006a] this synchronization is done with a global collective operation.

Our approach combines non-blocking one-sided messages with local synchronization as suggested by Kjolstad and Snir [2010]. A phase counter is assigned to each

ghosted `DistArray`. The use of a unique phase counter per array allows ghost updates on different arrays to proceed independently. This can be of use, for example, in a multigrid or adaptive mesh refinement algorithm in which different timesteps are used for coarser or finer grids. In each even-numbered phase the program computes on ghost data; in each odd-numbered phase ghost data are exchanged with neighboring places. A place may not advance more than one phase ahead of any neighboring place. A call to `sendGhostsLocal()` increments the phase for this place and then sends active messages to update ghost data at neighboring places. Each active message also sets a flag to notify the receiving place that data have arrived from a particular neighbor. The receiving place calls `waitForGhostsLocal()` to check that flags have been set for all neighbors before proceeding with the next computation phase.

Split-Phase Ghost Updates

The use of local synchronization allows phases to proceed with computation before neighboring places have received their ghost data; it also allows communication of ghost data to overlap with computation on local data at each place, as follows:

```
1  // at each place
2  for (i in 1..ITERS) {
3    sendGhostsLocal();
4    computeOnLocalData();
5    waitForGhostsLocal();
6    computeOnGhostData();
7  }
```

This approach is similar to the split-phase barrier in UPC, discussed in chapter 2.

Use of Active Messages

In the implementation of ghost updates, active messages are used to transfer and perform local layout of ghost data, and to ensure consistency of data for each phase of computation. In `sendGhostsLocal()`, each place sends messages to neighboring places. A conditional statement (`when`) ensures that the ghost data are not updated until the receiving place has entered the appropriate phase. After ghost data have been updated, a flag is set within an atomic block to indicate that the data have been received:

```
1  at(neighbor) async {
2    val mgr = localHandle().ghostManager;
3    when (mgr.currentPhase() == phase);
4    for (p in overlap) {
5      ghostData(p) = neighborData(p);
6    }
7    atomic
8      mgr.setNeighborReceived(sourcePlace);
9  }
```

In `waitForGhostsLocal()`, another conditional atomic block is used to wait until ghost data have been received from all neighboring places:

```
1  public def waitForGhostsLocal() {
2    when (allNeighborsReceived()) {
3      currentPhase++;
4      resetNeighborsReceived();
5    }
6  }
```

Support for ghost region updates in block-distributed arrays was added to `x10.regionarray.DistArray` in X10 version 2.5.2. Ghost region support will be added to the basic distributed array classes in the `x10.array` package in future versions of X10. Further enhancements are possible, for example, support for irregular distributions, and ghost regions with different (or zero) extent in different dimensions.

3.3.2.2 Evaluation of Ghost Updates

We performed a microbenchmark on *Vayu* to compare the scaling of the local synchronization implementation of ghost region updates with an alternative algorithm using a global barrier. This benchmark performs 10,000 ghost updates for a seven-point stencil over a large distributed three-dimensional array. The array size for one place is 100^3, and the array size is increased with the number of places (weak scaling), so that each place always holds one million double-precision values. Figure 3.10 shows the scaling measured for each update algorithm, which demonstrates the benefit of using only local synchronization as compared to a global barrier.

For 2–8 places, the difference between local and global synchronization is insignificant, as all communications (including the global barrier) take place within a single node. For more than eight places, the update time with a global barrier increases more rapidly than the time with only local synchronization. This is expected as the collective operation on which it is implemented scales as $\mathcal{O}(\log p)$ where p is the number of places, whereas other elements of the ghost update scale as $\mathcal{O}(1)$. These results show that the cost of updating ghost regions can be reduced by using local synchronization, rather than global synchronization.

3.4 Summary

This chapter proposed new features for the X10 programming language to support scientific application programming, and considered their efficient implementation on modern computer architectures. The co-design process followed in this work generated a number of insights regarding the language design that resulted in changes to improve performance and programmability for task parallelism, active messages and distributed arrays. The value of these changes will be demonstrated in the context of complete scientific application codes in chapters 4 and 5.

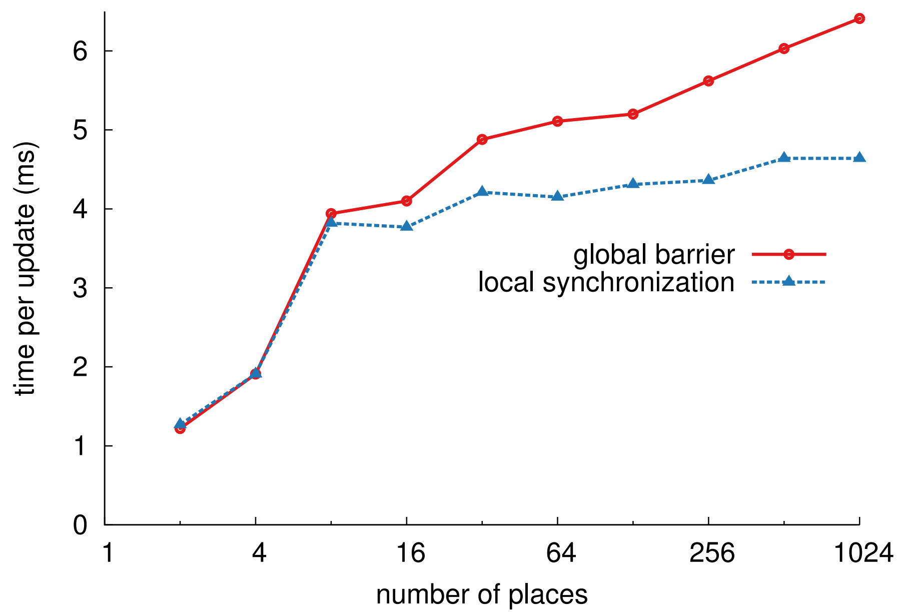


Figure 3.10: Ghost region update weak scaling on *Vayu*: global barrier vs. local synchronization (1M elements per place, 8 places per node).

Chapter 4

Electronic Structure Calculations Using X10

This chapter describes the application of the APGAS programming model to a quantum chemistry problem: the calculation of Hartree–Fock energy using the self-consistent field method (SCF). We implemented SCF using the resolution of the Coulomb operator (RO) approach previously described in chapter 2, entirely in the X10 programming language, exploiting both distributed parallelism between places and multithreaded parallelism within each place. During implementation, we considered the following questions:

- How can the novel RO method be structured to exploit parallelism?
- How does X10's work stealing runtime affect load balancing and data locality in the computation of auxiliary integrals (the key intermediate representation in RO)?
- How should the key data structures be distributed between X10 places?
- How do choices about data distribution affect load balancing between places?
- What benefits can be achieved through the use of optimized implementations of dense linear algebra operations?

Several of the improvements to the X10 programming language that were proposed in chapter 3 are demonstrated in the context of this application. The visualization of task locality in multithreaded execution is used to evaluate different schemes for generating activities within a place. Issues of data distribution are explored with regard to the *distributed array* data structure. The collective active messages **`finish / ateach`** are used to create and coordinate parallel activities across a large number of places. The `PlaceLocalHandle` and `WorkerLocalHandle` structures are used to support data replication and combining partial contributions to key matrices. The contributions of these improvements and the performance of the application are evaluated on representative parallel computing architectures.

Portions of this chapter have been previously published in Milthorpe et al. [2011], Limpanuparb et al. [2013] and Limpanuparb et al. [2014]. All source code included in this chapter as well as the benchmarks used in evaluation are distributed as part of the ANUChem suite of computational chemistry applications in X10 [ANUChem].

4.1 Implementation

Pumja Rasaayani (Sanskrit for *quantum chemistry*) is a complete Hartree–Fock implementation in the X10 programming language, developed by the present author in collaboration with Ganesh Venkateshwara and Taweetham Limpanuparb. The implementation is unique in that it uses the linear-scaling resolution of the Coulomb operator (RO) described in chapter 2 to reduce the computational complexity of Fock matrix construction. However, the challenges for parallelization are similar to standard SCF calculations where RO is not used.

As discussed in chapter 2, the resolution of the Coulomb and exchange operators for a set of N basis functions requires the computation of auxiliary integrals $(\mu\nu|nlm)$ for shell pairs μ, ν where $n \in \{0 \ldots \mathcal{N}'\}$ is the radial component and $l \in \{0 \ldots \mathcal{L}\}, m \in \{-l \ldots l\}$ are the angular components of the resolution. Constructing the Fock matrix requires a total of $N^2\mathcal{N}'(\mathcal{L}+1)^2$ auxiliary integrals; however, as the contributions for the different radial components of the resolution are independent, the integrals may be computed and their contributions to the Coulomb matrix J and the exchange matrix K accumulated in a loop over n, thereby reducing the storage required for integrals by a factor of $1/(\mathcal{N}'+1)$. Dividing by n (rather than by angular component l, m) allows the use of recurrence relations to efficiently compute the resolution for higher angular momenta [Limpanuparb et al., 2012].

1	For $n = 0 \ldots \mathcal{N}'$			
2	$A_{\mu,\nu lm}$	$\leftarrow$	$(\mu\nu\|nlm)$	Eq. (2.20)
3	// Coulomb matrix			
4	D^{lm}	$\leftarrow$	$P_{\mu\nu} \times A_{\mu\nu,lm}$	Eq. (2.22)
5	$J_{\mu\nu}$	$+=$	$D^{lm} \times A_{\mu\nu,lm}$	Eq. (2.21)
6	// Exchange matrix			
7	$B_{\nu lm,a}$	$\leftarrow$	$A_{\mu,\nu lm} \times C_{\mu,a}$	Eq. (2.24)
8	$K_{\mu\nu}$	$+=$	$B_{\mu,lma} \times B_{\nu,lma}$	Eq. (2.23)
9	Next n			

Figure 4.1: Pseudocode for construction of Coulomb and exchange matrices using resolution of the operator (from Limpanuparb et al. [2013])

In a previous paper, we presented a sequential implementation of RO as shown in the pseudocode in figure 4.1 [Limpanuparb et al., 2013]. For parallel implementation, it may be noted that while the elements of both the J matrix (lines 4–5) and the K matrix (lines 7–8) depend on the auxiliary integrals (line 2), J and K may be

computed independently of each other. Each $J_{\mu\nu}$ depends on all values of D^{lm} (a reduction dependency), which in turn depends on the auxiliary integrals A and the density matrix P. Each $K_{\mu\nu}$ depends on two blocks of $B_{\nu lm,a}$, which in turn depends on the auxiliary integrals A and the molecular orbital coefficients matrix C. The most expensive operations are those to produce the exchange matrix at lines 7 and 8, costing $\mathcal{O}(ON^2\mathcal{K})$ operations, but these can be formulated as matrix multiplications and efficiently computed using optimised (double-precision general matrix multiply (DGEMM)) library calls. The memory requirements for matrices A and B are $\mathcal{O}(N^2(\mathcal{L}+1)^2)$ and $\mathcal{O}(ON(\mathcal{L}+1)^2)$ respectively. It was shown that the computational cost of Hartree–Fock J and K matrix calculation using high quality basis sets can be significantly reduced by using the resolution. RO algorithm scales only quadratically with respect to basis set size (for a fixed molecule) and works best for compact globular molecules [Limpanuparb et al., 2013], for which traditional cut-off strategies [Häser and Ahlrichs, 1989] are ineffective.

In figure 4.1, the auxiliary integrals are calculated and stored in a dense matrix $A_{\mu,\nu lm}$ (line 2). In comparison to the previous paper, we insert an additional step 2a which stores a copy of the integrals in sparse format as a ragged array $A_{\mu\nu,lm}$, with as much storage as required for the maximum angular momentum of each shell pair. This dual storage approach allows the contribution of an entire shell to matrix B (line 7) to be computed using an efficient DGEMM operation, while the contraction of integrals with the density matrix (line 4) reads integral values with unit stride in memory.

The actual computation of the integrals is done by a native call to an optimized C++ function, which calculates a complete class of integrals $(\mu\nu|nlm)$ for a given shell pair **A**, **B**. This function has the following signature:

```
1  void Genclass(
2    // angular momenta
3      int angA,               int angB,
4    // coordinates of centers
5      const double *A,      const double *B,
6    // orbital exponents
7      const double *zetaA, const double *zetaB,
8    // contraction coefficients
9      const double *conA,  const double *conB,
10   // degrees of contraction
11     int dconA,              int dconB,
12   // radial component of resolution
13     int n,
14   // spherical harmonics for given n
15     double *Ylm,
16   // [out] auxiliary integrals
17     double *aux
18 );
```

After return of a call to `GenClass`, the array `aux` contains the indexed set of integrals $A_{\mu\nu lm}$ for the given n.

4.1.1 Parallelizing the Resolution of the Operator

The computations of the auxiliary integrals and their contributions to the J and K matrices permit multiple levels of parallelism. The most natural levels at which to divide the computation are by shell (a block of μ) or shell pair (a block of $\mu\nu$), and radial component n.

When decomposing by shell, the J and K matrix contributions for each block of μ depend on the full sets D^{lm} and $B_{\nu lma}$, which must therefore also be reduced across all places involved in computing integrals for a given n. The auxiliary integrals $A_{\mu,\nu lm}$ and intermediate matrix $B_{\mu,lma}$ may be divided between places, to allow larger problems to be treated by using the available memory on multiple cluster nodes. Parallelism within a place is also readily exposed by dividing by shell; as each shell contributes to a unique block of both J and K matrices, threads may proceed independently without the need for mutual exclusion. Integral calculation may be further divided by shell pair (a block of $\mu\nu$) to increase the available parallelism; however, this means that each activity at line 7 in figure 4.1 contributes to a very small block of B — possibly as small as one element. The coarser division by shell allows the computation of a larger block of B to be performed by a call to an efficient BLAS DGEMM operation, which is the approach used in this implementation.

When decomposing by the radial component, each value of n requires all spherical harmonics $Y_{lm}^{\mu\nu}$ (2.19), so distributing different values of n to different processes requires that the full set of $Y_{lm}^{\mu\nu}$ must be either replicated or recomputed at each place. As the full matrices $A_{\mu\nu lm}$, $B_{\nu lma}$ are required for each value of n, there is no reduction in the memory requirement per node, and therefore dividing by radial component does not by itself permit the treatment of a larger problem than would fit in the memory of a single cluster node. Distribution of n, however, may offer time saving when the SCF requires many cycles and auxiliary integrals are not recalculated after the first cycle. The current implementation uses only shell decomposition for place parallelism.

The time required to compute auxiliary integrals depends on the angular momenta of the basis functions, and may differ widely between shells. Figure 4.2 shows the time to compute auxiliary integrals for the different shells of a cluster of 10 water molecules using the cc-pVQZ basis set. The time required to compute a single shell varies from 11 ms to more than 70 ms. This presents a load balancing problem as it is necessary to divide the set of integrals into evenly sized portions for computation by different processing elements. A static load balancing approach - carefully mapping integrals to processing elements so that each processing element performs roughly the same amount of computational work - has been successfully used for integral evaluation on GPUs [Ufimtsev and Martínez, 2008]. Even so, static schemes cannot account for unpredictable changes in the compute capacity of different cores that may arise when, for example, the frequency of a core is automatically reduced in response to local environmental factors. Alternatively, load may be balanced dynamically by ordering integral calculations in a centralized task queue [Asadchev and Gordon, 2012]; such an approach has previously been considered for use with X10 [Shet et al.,

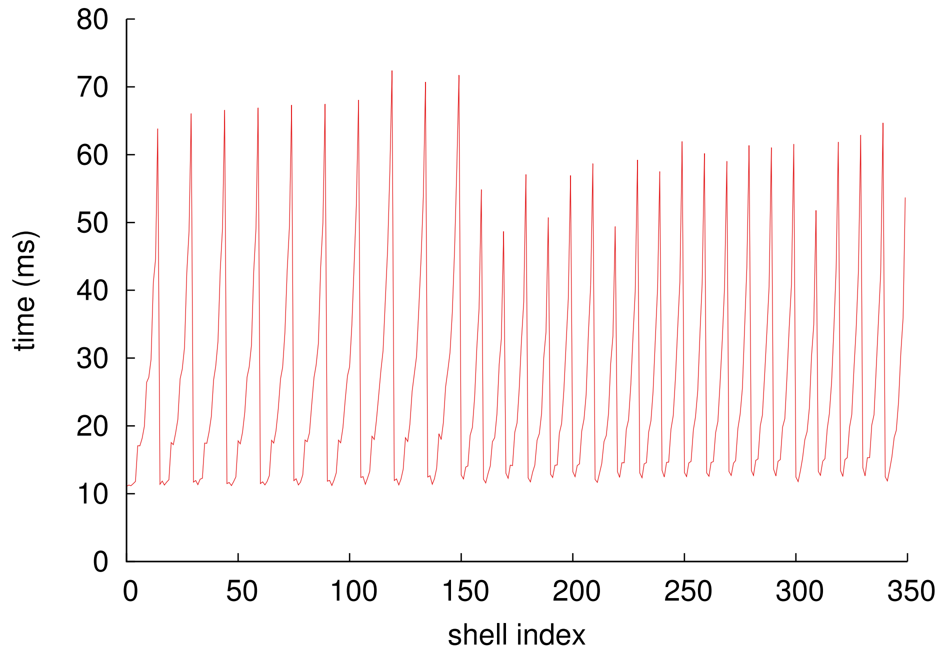


Figure 4.2: Variation in time to compute auxiliary integrals for different shells on *Raijin* ($[H_2O]_{10}$, cc-pVQZ, $\mathcal{N}' = 9, \mathcal{L} = 16$)

2008; Milthorpe et al., 2011]. X10 provides a partial solution to this problem by allowing worker threads within a place to dynamically balance load using work stealing, without the need for a centralized task queue.

In §4.1.2 we explore an alternative solution which makes use of X10's work stealing runtime for dynamic load balancing within a place, without the need for a centralized task queue. In §4.1.3 we describe the distribution of key data structures needed for multi-place computation. Finally, in §4.1.4 we consider the load imbalance between places due to the choice of data distribution.

4.1.2 Auxiliary Integral Calculation with a Work Stealing Runtime

Although work stealing can be an effective way of load balancing parallel activities between worker threads, it also raises three potential problems: i) how to avoid synchronization so as to allow activities to compute independently; ii) the overhead of creating and stealing activities; and iii) the impact of stealing patterns on data locality. The following subsections consider each problem in the context of auxiliary integral calculation.

4.1.2.1 Use of Worker-Local Data to Avoid Synchronization

The `WorkerLocalHandle` class described in chapter 3 allows parallel activities to compute contributions to the J and K matrices independently of other activities. Figure 4.3

shows the use of a `WorkerLocalHandle` to allocate memory for a separate set of auxiliary integrals for each worker thread. The parallel loop on line 5 starts a separate

```
1  val aux_wlh = new WorkerLocalHandle[Rail[Double]](
2      ()=> new Rail[Double](maxam1*N*roK) );
3  val dlm_wlh = new WorkerLocalHandle[Rail[Double]](
4      ()=> new Rail[Double](roK) );
5  finish for (shell in shells) async {
6    val aux = aux_wlh();
7    val dlm = dlm_wlh();
8    for (shellPair in shell.pairs) {
9      Genclass(shellPair, ..., aux);
10     for ([mu,nu] in shell.basisFunctions) {
11       for (l in 0..roL) {
12         for (m in -l..l) {
13           dlm(l,m) += density(mu,nu) * aux(mu,nu,l,m);
14         }
15       }
16     }
17   }
18   DGEMM(mos, aux, b); // Single-threaded
19 }
20 dlm_wlh.reduceLocal(dlm_complete, ...);
```

Figure 4.3: X10 code to compute auxiliary integrals, D^{lm} and B using `WorkerLocalHandle`

activity (**`async`**) for each shell, and waits for all activities to terminate (**`finish`**); the activities are scheduled for execution on the available worker threads. The worker threads also use a `WorkerLocalHandle` to store partial contributions to D^{lm}, which are summed together after the computation of all auxiliary integrals for a given n. Thus each iteration of the parallel loop in figure 4.3 is independent and synchronization is only required in the reduction of D^{lm} on the final line. The improvements to `WorkerLocalHandle` described in § 3.1.1 ensure that the minimum number of local copies are required, and provide a simple and correct reduction over all worker-local data.

4.1.2.2 Overhead of Activity Management

For efficient scheduling, work stealing relies on over-decomposition — a division of work into significantly more activities than there are units of hardware parallelism. It is possible that the cost of creating many activities and scheduling them for execution by worker threads could outweigh the benefit gained by balancing load. To evaluate the costs and benefits for this application requires examination of the pattern of activity creation and stealing exhibited at runtime.

For the basic parallel loop shown in figure 4.3 (line 5), the worker thread that executes the enclosing **`finish`** statement (the *master* thread) creates a separate activity for each iteration of the loop. When an activity is created, if any threads are currently

idle then the activity is dealt to one of the idle threads. Otherwise the activity is placed on the deque of the worker thread that created it. The majority of activities are therefore placed on the deque of a single thread, to be stolen by other threads. In other words, the steal ratio — the proportion of activities that are stolen rather than executed by the worker which created them — is high.

To evaluate the benefit of work stealing in this application, we compare it with the static load balancing approach, in which a single activity is created per thread. Figure 4.4 shows code for such a static load balancing approach using a cyclic decomposition of shells.

```
1  finish for (th in 0..(Runtime.NTHREADS-1)) async {
2    for (shellIdx = th; shellIdx < shells.size; shellIdx += maxTh) {
3      Genclass(...);
4    }
5  }
```

Figure 4.4: Parallel loop to compute auxiliary integrals: single activity per thread with cyclic decomposition

Using a single long-running activity per worker thread eliminates the overhead of activity management, however, there may be a load imbalance between worker threads due to the differing times to compute shell pairs as shown in figure 4.2.

4.1.2.3 Optimizing Auxiliary Integral Calculations for Locality

For large problem sizes it is not possible to store all auxiliary integrals in cache. For example, for a cluster of 5 water molecules using the cc-pVQZ basis set where $N = 700, \mathcal{L} = 13$, there are $700^2 \times (13+1)^2 \approx 96\,\text{M}$ auxiliary integrals requiring 733 MiB, which is around two orders of magnitude larger than a typical last-level cache. To achieve performance it is necessary to make efficient use of cache.

The most natural order in which to store the auxiliary integrals (using either sparse or dense matrix representation) is by shell pair μ, ν. Thus there is a relationship between `shellIdx` and the memory locations to which the corresponding auxiliary integrals must be written. Using the basic parallel loop shown in figure 4.3, the order of stealing is effectively random, which means that there is no relationship between the loop index `shellIdx` for an activity and the worker thread which executes it. In comparison, the cyclic decomposition in figure 4.4 means that each thread's accesses are roughly evenly distributed throughout memory, in small contiguous blocks separated by irregular stride.

A single activity per worker thread may also use a block decomposition, as shown in figure 4.5. This divides the shell pairs into contiguous blocks of approximately equal size. The block decomposition has the additional advantage of locality: each worker thread writes to a compact portion of the auxiliary integral matrix $A_{\mu,\nu lm}$, whereas with the cyclic decomposition each thread writes to widely separated portions of the matrix.

```
1  val chunk = shells.size / Runtime.NTHREADS;
2  val remainder = shells.size % Runtime.NTHREADS;
3  finish for (th in 0..(Runtime.NTHREADS-1)) async {
4    val start = th < remainder ? ((chunk+1) * th)
5                               : (remainder + chunk*th);
6    val end =  (th < remainder ? ((chunk+1) * (th+1))
7                               : (remainder + chunk*(th+1))) - 1;
8    for (shellIdx in start..end) {
9      Genclass(...);
10   }
11 }
```

Figure 4.5: Parallel loop to compute auxiliary integrals: single activity per thread with block decomposition

The basic parallel loop entails a high steal rate and therefore stealing overhead, whereas the static load balancing approaches may lead to load imbalance. Ideally, we would prefer a loop decomposition that provides load balancing with a low steal rate, while also maintaining good locality between the tasks processed by each worker thread. The divide-and-conquer loop transformation introduced in chapter 3 promises such a combination. In this approach, the loop is recursively bisected into approximately equal ranges, with each range constituting an activity. Bisection continues until a certain minimum grain size is reached. Figure 4.6 shows example code for this transformation applied to the loop over shell pairs.

With the recursive bisection loop transformation, if a worker thread's deque contains any activities, then the activity at the bottom of the deque will represent at least half of the loop range held by that worker. Thus idle workers tend to steal large contiguous chunks of the loop range, preserving locality. A further advantage is that activities are executed in order of shell index, rather than in reverse order as for the basic parallel loop. The bisection approach is not without cost: for a loop of N iterations, an additional $\log_2(N/\text{grainSize})$ activities are created to divide the work.

4.1.3 Distributed and Replicated Data Structures

When implementing the SCF for execution on a distributed architecture, an important question is how to distribute the key matrices. Certain matrices such as the density matrix and molecular orbital coefficients must be accessible to all places, as their elements are used in the calculation of widely separated elements of the Fock matrix. If the size of the system is small enough, it is possible to replicate these matrices so that a copy is held at every place. In X10, this may be done using a `PlaceLocalHandle` (see §3.1.1).

Figure 4.7 shows the high-level structure of the X10 code to compute the Fock matrix using RO. The density and molecular orbital matrices are updated each iteration at Place 0, after which they are replicated to all other places (line 4) using the broadcast active message that was described in chapter 3. For large problems,

```
1  struct RecursiveBisection1D(start:Long, end:Long, grainSize:Long){
2    public def this(start:Long, end:Long, grainSize:Long) {
3      property(start, end, grainSize);
4    }
5
6    public def execute(body:(idx:Long)=> void) {
7      if ((end-start) > grainSize) {
8        val secondHalf =
9          RecursiveBisection1D((start+end)/2L, end, grainSize);
10       async secondHalf.execute(body);
11       val firstHalf =
12         RecursiveBisection1D(start, (start+end)/2L, grainSize);
13       firstHalf.execute(body);
14     } else {
15       for (i in start..(end-1)) {
16         body(i);
17       }
18     }
19   }
20 }
21
22 finish RecursiveBisection1D(0, shells.size, grainSize).execute(
23   (shellIdx:Long)=> {
24     ... // Line 6--18 in Figure 3
25 }
26 );
```

Figure 4.6: Parallel loop to compute auxiliary integrals: recursive bisection transformation

```
 1 public class ROFockMethod {
 2   ...
 3   public def compute(density:Density, mos:MolecularOrbitals) {
 4     finish ateach(place in PlaceGroup.WORLD) {
 5       for (ron in 0n..roN) {
 6         computeAuxBDlm(density, mos, ron,
 7                        /*output*/ auxJ, bMat, dlm);
 8         computeK(ron, bMat,
 9                        /*output*/ localK);
10         computeJ(ron, auxJ, dlm,
11                        /*output*/ localJ);
12         Team.WORLD.barrier();
13       }
14       // gather J, K to place 0
15     }
16   }
17   ...
18 }
```

Figure 4.7: High level structure of X10 code to compute Fock matrix, showing broadcast of density and molecular orbital matrices using **finish/ateach**

computing these matrices (density and MO) entirely at Place 0 may present an undesirable sequential bottleneck; however for the problems considered in this chapter the cost of this step is small relative to other parts of the computation.

Other matrices are too large to replicate in this way, and must be distributed between places; care must be taken to distribute them so as to minimize data movement between stages of the computation. As previously mentioned, for large molecules it is not possible to replicate the auxiliary integrals $A_{\mu\nu,lm}$. The auxiliary integral matrix is therefore divided between places into blocks according to shell index μ.

4.1.4 Load Balancing Between Places

The computational cost attributable to a given shell is estimated as being proportional to the total angular momentum of all associated shell pairs; shells are divided so that total angular momentum is approximately even between places. If possible, integrals for an entire shell are computed at a single place. Where assigning a whole shell to one place would lead to a load imbalance, the basis functions associated with the shell are divided between two or more places. The J matrix is distributed in the same way, so that contributions to J at each place may be calculated entirely from integrals held at that place.

Given the data distribution scheme described above, there are two different measurements of computational cost at each place. The first is the Fock matrix cost $\text{cost}_{\text{Fock}}$, which depends on the size of the portion of the J and K matrices and therefore the Fock matrix that are computed at each place; this is determined simply by

the number of shells M_p assigned to place p:

$$\text{cost}_{\text{Fock}}(p) = (M_p)^3. \tag{4.1}$$

The second measurement is the auxiliary integral cost cost_{aux}, which is the total angular momentum of all shell pairs for which auxiliary integrals are computed at place p:

$$\text{cost}_{\text{aux}}(p) = \sum_{\mu=1}^{M_p} \sum_{\nu} \text{ang}(\mu, \nu). \tag{4.2}$$

From these costs, two different measurements of load imbalance can be calculated: the Fock matrix load imbalance

$$L_{\text{Fock}} = (\max_p \text{cost}_{\text{Fock}}(p)) / (\sum_{p=0}^{P-1} (\text{cost}_{\text{Fock}}(p)) / P), \tag{4.3}$$

and the auxiliary integral load imbalance

$$L_{\text{aux}} = (\max_p \text{cost}_{\text{aux}}(p)) / (\sum_{p=0}^{P-1} (\text{cost}_{\text{aux}}(p)) / P). \tag{4.4}$$

Assuming a perfectly parallel code with no sequential sections or parallel overhead, the load imbalance is the inverse of parallel efficiency. For example, an imbalance of 4 would mean a parallel efficiency of 25% and an imbalance of 1 would mean perfect parallel efficiency.

4.1.5 Dense Linear Algebra Using the X10 Global Matrix Library

The computation of K matrix contributions from auxiliary integrals can be cast in the form of dense linear algebra operations, which enables the use of a highly-optimized BLAS implementation. The contraction of auxiliary integrals with molecular orbitals (step 7 in figure 4.1) for a given n is performed using a call to DGEMM, which uses a single thread to allow multiple such calls in parallel for different shells, each of which creates a unique block of the matrix $B_{\nu lma}$. Once B has been computed in full, the contribution to matrix K (step 8 in figure 4.1) may be computed using a symmetric rank-K update. For a single place, all of B is held locally, which allows the use of the BLAS double-precision symmetric rank-K update (DSYRK) operation. For multiple places, B is divided into blocks of rows, meaning that a distributed rank-K update is required.

Matrices B and K (as well as J) are represented using the `DistDenseMatrix` class from the X10 Global Matrix Library (GML). In a `DistDenseMatrix`, the matrix is block-distributed between places; each place maintains its own local block as a `DenseMatrix`, and also holds a `Grid` which defines the extent of the local block held at each place in the distribution. For our purposes, matrix B is divided into blocks of rows according to the values of μ for which that place has computed auxiliary integrals. The block

of B held by each place must include a square of size approximately N/p on the diagonal. From this local square block, the corresponding upper-left triangle block of K can be computed using DSYRK. The remaining off-diagonal blocks of K are computed at each place using DGEMM on a combination of local and non-local data.

Figure 4.8 shows the pattern of distributed computation of blocks of the K matrix for 1–4 places. For odd numbers of places, all DGEMM operations compute square blocks of K. For even numbers of places, to ensure load balance the last block in each row is divided between pairs of places, therefore the final DGEMM for each place computes a rectangular block of K.

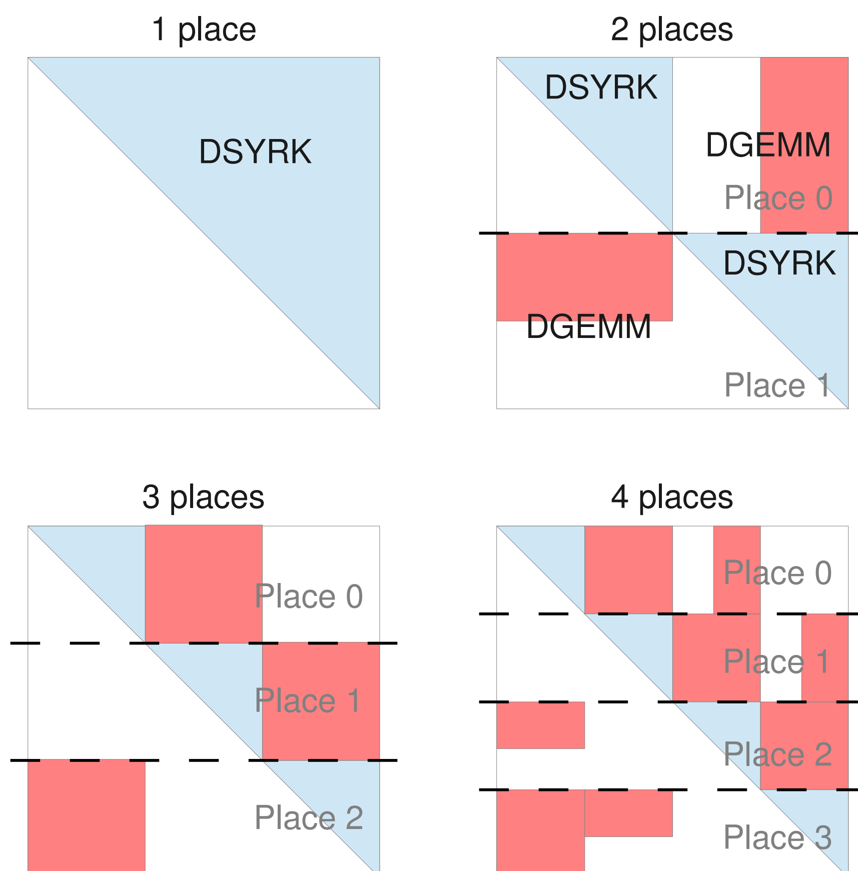


Figure 4.8: RO contributions to K matrix: block pattern of distributed computation for different numbers of places

After each place has computed its partial contributions to J and K, these matrices are gathered to Place 0 for further processing - for example, to compute long-range Coulomb and Exchange energies, or to compute new molecular orbital coefficients and density matrices in a full SCF calculation. The transfer of each partial contribution is performed using the `Rail.asyncCopy` method which asynchronously transfers array data to a remote place. While this method is efficient for small numbers of

places, it is not expected to scale to larger numbers of places. A more scalable solution would be a tree-based collective communication similar to `MPI_Gatherv`; efforts are currently underway to provide such functionality in the X10 Team API.

An alternative solution would be not to gather the J and K matrices at all, but to leave them distributed between places. The various matrix operations required for energy or SCF calculations would therefore be implemented as distributed linear algebra operations.

4.2 Evaluation

To evaluate the X10 implementation of SCF using resolution of the Coulomb operator (RO) (*Pumja Rasaayani*), its performance will first be compared against that of a conventional SCF calculation using Q-Chem version 4.0.0.1 [Shao et al., 2013] on *Raijin*.[1] Shared-memory scaling will then be considered, with particular regard to the effects on locality and performance of the different methods for dividing loops for integral and J matrix calculation previously described in §4.1.2. Finally, distributed-memory scaling will be assessed with regard to load balance and communication required between places. The Intel Math Kernel Library version 12.1.9.293 implementation of the BLAS library was used for the X10 code.

4.2.1 Single-Threaded Performance

A single SCF cycle was performed for calculation of long-range energy using Ewald partition parameter $\omega = 0.3$ and accuracy threshold THRESH $= 6$ for various molecules with basis sets of different quality. Molecular orbitals from diagonalization of the core Hamiltonian and Cartesian orbitals were used for all calculations in this section. Table 4.1 shows the time to compute Fock matrix, including auxiliary integral, J and K matrix calculations, for various molecules on a single core of *Raijin*. The test cases represent typical requirements for simulations of biological systems: clusters of five to twenty water molecules [Maheshwary et al., 2001], and one-dimensional (chain-like) and three-dimensional (globular) alanine polypeptides [DiStasio et al., 2005]. In addition to computation time, the table reports characteristics of each calculation including number of occupied orbitals O, number of basis functions N, molecular radius R, resolution radial truncation $\mathcal{N}$, and angular truncation $\mathcal{L}$. For each molecule, basis sets are ordered down the table from smallest to largest number of basis functions. To compare the performance of *Pumja Rasaayani* against conventional SCF calculation, the time for Fock matrix calculation using Q-Chem is also reported.[2]

As discussed earlier in §2.5.2, the RO computational cost is $\mathcal{O}(ON^2\mathcal{K})$. For each molecular system in table 4.1, we can treat O and $\mathcal{K}$ as constants. Therefore, we

[1] As there is no other complete SCF implementation using resolution of the Coulomb operator (RO), it is necessary to compare against a conventional SCF implementation.

[2] The reported time for Q-Chem is the "AOints" time for the full Coulomb operator. This is an approximation to the conventional long-range J and K matrix calculation time, as the number of integrals required to be calculated is approximately the same.

Table 4.1: Time to compute Fock matrix components for different molecules and basis sets using RO on *Raijin* (one thread)

							SCF time (s)*				
							RO				Q-Chem
Molecule	**Basis set**	O	N	R	$\mathcal{N}'$	$\mathcal{L}$	**Aux**	**J**	**K**	**Total**	**Total**
$(H_2O)_5$	cc-pVDZ	25	125	16.19	9	13	0.71	0.02	0.08	0.81	0.73
"	cc-pVTZ	"	325	."	"	"	3.67	0.10	0.36	4.15	7.99
"	cc-pVQZ	"	700	."	"	"	16.8	0.40	1.46	18.7	99.1
$(H_2O)_{10}$	cc-pVDZ	50	250	18.97	9	16	3.62	0.09	0.60	4.31	5.03
"	cc-pVTZ	"	650	"	"	"	20.5	0.41	3.31	24.2	60.3
"	cc-pVQZ	"	1400	"	"	"	90.8	1.51	13.8	106.4	824.0
$(H_2O)_{20}$	cc-pVDZ	50	500	31.07	14	25	51.1	0.77	14.5	66.3	27.2
"	cc-pVTZ	"	1300	"	"	"	314.1	3.45	81.0	398.7	346.0
"	cc-pVQZ	"	2800	"	"	"	1768	13.0	355.0	2138	5480
1D-alanine$_4$	6-311G	77	326	34.52	15	27	23.4	0.47	6.36	30.27	6.32
"	cc-pVDZ	"	410	"	"	"	43.4	0.69	9.50	53.63	27.8
"	cc-pVTZ	"	1030	"	"	"	280.0	3.51	53.0	336.7	377.0
"	cc-pVQZ	"	2170	"	"	"	1317	19.8	243.47	1581	6530
3D-alanine$_4$	6-311G	77	326	22.91	11	19	9.88	0.24	2.44	12.56	9.09
"	cc-pVDZ	"	410	"	"	"	18.4	0.35	3.68	22.47	37.9
"	cc-pVTZ	"	1030	"	"	"	95.2	1.61	19.5	116.4	491.0
"	cc-pVQZ	"	2170	"	"	"	413.3	5.97	80.0	500.0	8700
1D-alanine$_8$	6-311G	153	646	61.64	24	46	502.2	4.12	188.3	694.6	27.8
"	cc-pVDZ	"	810	"	"	"	843.9	6.92	278.4	1129	119.0
"	cc-pVTZ	"	2030	"	"	"		†		†	1800
"	cc-pVQZ	"	4270	"	"	"		†		†	30600
3D-alanine$_8$	6-311G	153	646	30.56	13	24	82.6	1.13	30.07	113.8	55.8
"	cc-pVDZ	"	810	"	"	"	142.4	1.54	44.9	188.9	208.0
"	cc-pVTZ	"	2030	"	"	"	946.7	7.49	261.7	1217	3070
"	cc-pVQZ	"	4270	"	"	"		†		†	‡

* We used Q-CHEM default THRESH of 8 and RO default THRESH of 6 as it has been shown previously [Limpanuparb et al., 2013] that these default setup give comparable level of accuracy. To confirm the reliability of our RO method, we compute $\epsilon = -\log_{10}|E/E^{\mathrm{REF}} - 1|$ for 1D-alanine$_4$/6-311G, 1D-alanine$_4$/cc-pVDZ, 3D-alanine$_8$/6-311G, 3D-alanine$_{16}$/6-311G, $(H_2O)_5$/cc-pVQZ, $(H_2O)_{10}$/cc-pVTZ and $(H_2O)_{20}$/cc-pVDZ. The ϵ_J are 8.56, 8.87, 8.39, 8.32, 9.15, 7.92, 8.80 and ϵ_K are 7.39, 7.41, 6.94, 6.65, 8.51, 7.08, 7.60, respectively. These are in line with the earlier result [Limpanuparb et al., 2013].

† Fock matrix for 1D-alanine$_8$ with cc-pVTZ and cc-pVQZ basis sets, and 3D-alanine$_8$ with cc-pVQZ could not be computed using RO on a single place due to lack of memory.

‡ Computation failed with Q-Chem due to a numerical stability problem with 'negative overlap matrix' reported.

observe that the computational time for RO increases approximately quadratically with the number of basis functions N, as expected. Q-Chem computation time rises much more rapidly, as conventional calculation is quartic in the number of basis functions for a fixed molecule.

By comparison, for a fixed basis set, the relationship between system size and computation time is not clear for either RO or conventional methods. In table 4.1, the observed increase in computation time for both conventional and RO methods is more than quadratic. Doubling the number of atoms (e.g. from $(H_2O)_5$ to $(H_2O)_{10}$) doubles both the number of occupied orbitals O and the number of basis functions N. It also tends to increase the resolution parameters $\mathcal{K}$ which depends on the molecule radius. However, the integral screening mentioned earlier may substantially reduce the actual computational cost when there are more atoms in the molecules. The conventional approach which is based on four-center two-electron integrals are likely to benefit from screening more than the RO approach which is based on three-center overlap integral. We therefore elect to use dense linear algebra operations for K matrix computation for reasons of efficient implementation.

A comparison of computation times for 1D- versus 3D-alanine$_4$ shows that RO is more effective for 3D molecules, because the required values of $\mathcal{N}'$ and $\mathcal{L}$ increase with the molecular radius. This is in contrast to traditional cutoff strategies, which are more effective for 1D chain-like molecules. RO's effectiveness for 3D molecules is a major advantage for biological applications, where 3D structures are the norm.

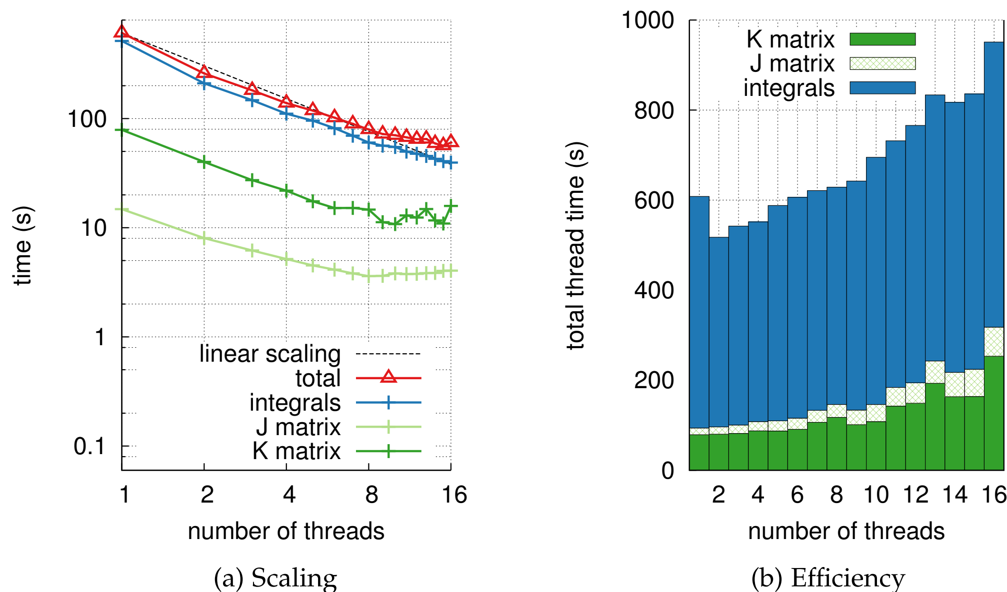
For all molecules tested, RO is slower than Q-Chem with small, low-quality basis sets, and faster than Q-Chem with large, high-quality basis sets due to its superior scaling with the number of basis functions.

4.2.2 Shared-Memory Scaling

We now consider the parallel speedup achievable by using multiple cores of a shared memory system with particular focus on the method used to divide auxiliary integral and J matrix calculations for parallel execution.

To measure parallel scaling, we selected a single test case from table 4.1: the 3D-alanine$_4$ polypeptide with the cc-pVQZ basis set. Figure 4.9(a) shows the scaling with number of threads of the major components of Fock matrix construction for RO long-range energy calculation on a single node of *Raijin*, using the basic parallel loop shown in figure 4.3. Figure 4.9(b) presents the same data in terms of parallel efficiency, showing the total thread time (elapsed time × number of threads). A component that exhibits perfect linear scaling shows constant total thread time, while an increase in total thread time represents a loss of parallel efficiency.

In figure 4.9, total Fock matrix construction time reduces substantially from 608.9 s on one thread to a minimum of 56.5 s on 15 threads, increasing slightly to 60.3 s on 16 threads. A node of *Raijin* has 16 physical cores, so ideal scaling would see linear speedup from 1 to 16 threads. Measured speedup reduces substantially above 8 threads; this appears to be largely due to poor scaling of K matrix computation. While the time for K matrix computation drops from 78.9 s on a single thread to



(a) Scaling (b) Efficiency

Figure 4.9: Multithreaded component scaling and efficiency of RO long range energy calculation on *Raijin* with basic parallel loops for integral and J matrix calculation (1–16 threads, 3D-alanine$_4$, cc-pVQZ, $\mathcal{N}' = 11, \mathcal{L} = 19$).

10.8 s on 10 threads, it increases again to 15.8 s on 16 threads. The bulk of K matrix computation for a single place is a call to a multithreaded BLAS DSYRK operation with input matrices of dimension 2170 $\times$ 2170. The increase in computation time suggests that the multithreaded implementation of DSYRK may be unsuitable for use on multiple sockets of *Raijin*. Therefore we restricted all subsequent runs to 8 threads per place, and for multi-place tests we ran two places per node, one bound to each socket. J matrix computation time also increases above 8 threads and total thread time increases substantially from 1 to 16 threads, which equates to a drop in parallel efficiency to just 22.8%. The poor scaling of J matrix computation is due to the high overhead of activity management relative to the cost of computation, as described in §4.1.2.2. Auxiliary integral computation scales much better, with computation time reducing steadily from 1 to 16 threads.

We next consider the overhead associated with work stealing. We measured the performance of the static load balancing approaches using either block or cyclic division of the auxiliary and J loops, as well as the recursive bisection loop transformation. Table 4.2 compares component timings for Fock matrix construction on a single socket of *Raijin* (8 cores/threads) for the four different methods of dividing the integral and J matrix loops that were described in §4.1.2.

Auxiliary integral computation is fastest using the basic parallel loop; followed by recursive bisection, then the static partitioning approaches. The J matrix computation is slowest using the basic parallel loop; block static partitioning is slightly faster, due to the reduced cost of stealing; cyclic partitioning is twice as fast as the basic loop, due

Table 4.2: Multithreaded component scaling of RO long range energy calculation on *Raijin* with different methods of dividing integral and J matrix loops between worker threads (8 threads, 3D-alanine$_4$, cc-pVQZ, $\mathcal{N}' = 11, \mathcal{L} = 19$)

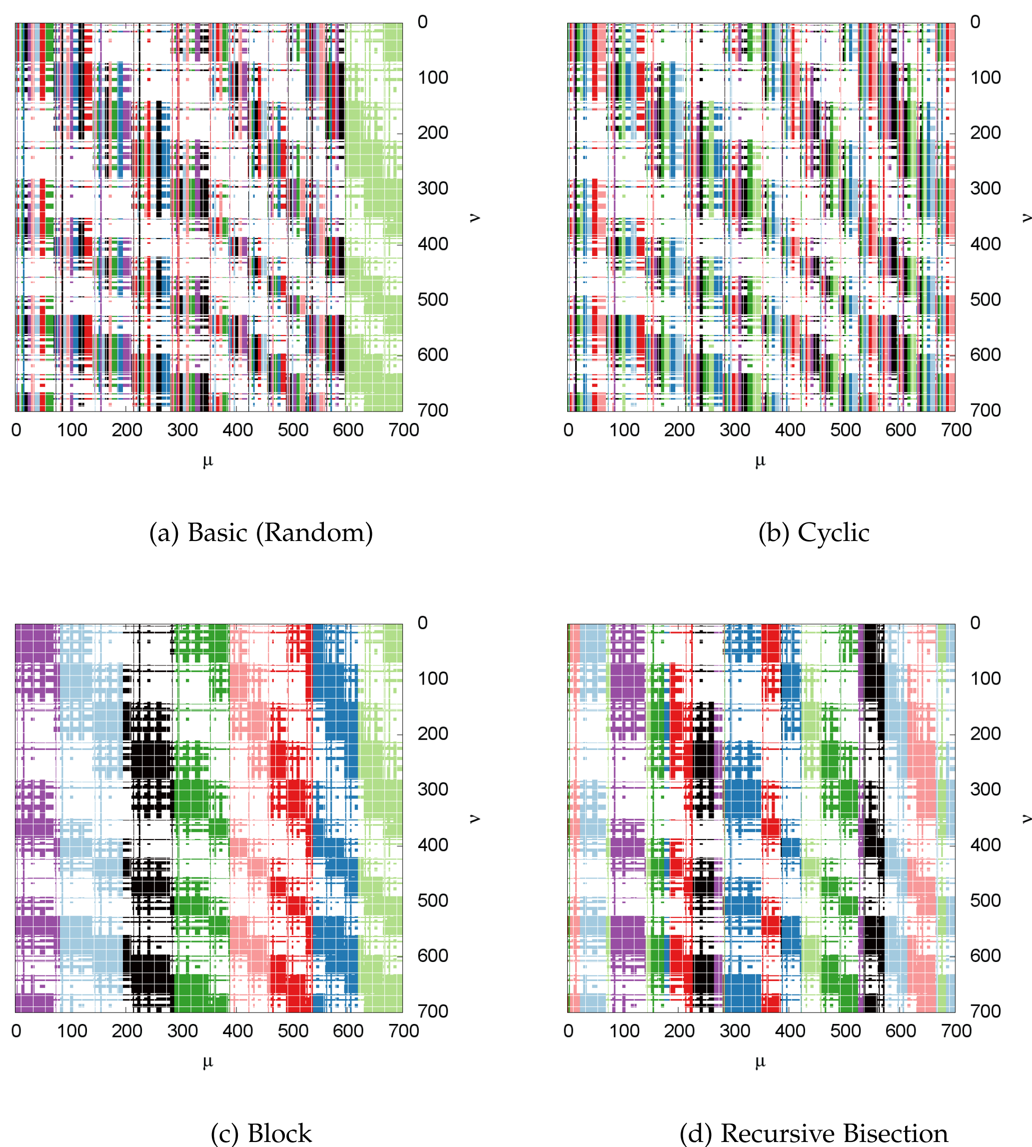
	time (s)			
component	**basic** (Figure 4.3)	**cyclic** (Figure 4.4)	**block** (Figure 4.5)	**bisection** (Figure 4.6)
Auxiliary integrals	60.35	69.51	68.35	61.24
J matrix	3.60	2.35	2.81	1.13
Total	**63.95**	**71.86**	**71.16**	**62.37**

to improved load balance between worker threads. Recursive bisection was by far the fastest approach for J matrix calculation, more than three times faster than the basic parallel loop. Overall, recursive bisection is the fastest; both dynamic partitioning approaches (bisection and basic) are significantly faster than the static partitioning approaches (cyclic and block).

To assist in understanding the performance of the different loop partitioning approaches, the locality of the activities executed during auxiliary integral computation was recorded for visualization using the approach presented in chapter 3. Figure 4.10(a) shows the activities to compute integrals for functions μ, ν over 8 worker threads using the basic parallel loop shown above, for a smaller problem size of five water molecules with the cc-pVQZ basis set. Activities are plotted in different colors according to which worker thread executed the activity. From the plot it is apparent that the activities for higher values of μ are all executed by the master thread, while the other activities are randomly distributed between the remaining threads. Figure 4.10(b) shows that with the cyclic decomposition, activities are dealt evenly to all threads, while figure 4.10(c) shows that with the block decomposition, each thread receives a roughly even-sized contiguous portion of μ, ν. Figure 4.10(d) shows that with recursive bisection, the bulk of the activities are divided into large, irregularly-sized, contiguous chunks. For this problem, both the cyclic decomposition and recursive bisection approaches result in even distribution of work, however, only recursive bisection also results in good locality of activities, which explains its superior performance.

We now return to the scaling and efficiency of multithreaded Fock matrix calculation for the same polyalanine system previously measured in figure 4.9. Figures 4.11(a) and 4.11(b) display the parallel scaling and efficiency achieved with recursive bisection of loops to compute auxiliary integrals and J matrix contributions.

The overall speedup is improved; the total time on 16 threads is 58.8 s in figure 4.11(a) using recursive bisection compared to 60.3 s in figure 4.9(a) with the basic parallel loop. The greatest difference is in J matrix calculation; the increase in total thread time (loss of parallel efficiency) is much smaller with the recursive bisection



(a) Basic (Random)

(b) Cyclic

(c) Block

(d) Recursive Bisection

Figure 4.10: Locality of auxiliary integral calculation on *Raijin* with different methods of dividing integral loop between worker threads (8 threads, $[H_2O]_5$, cc-pVQZ, $\mathcal{N}' = 9, \mathcal{L} = 13$). Activities executed by each worker thread are shown in a different color.

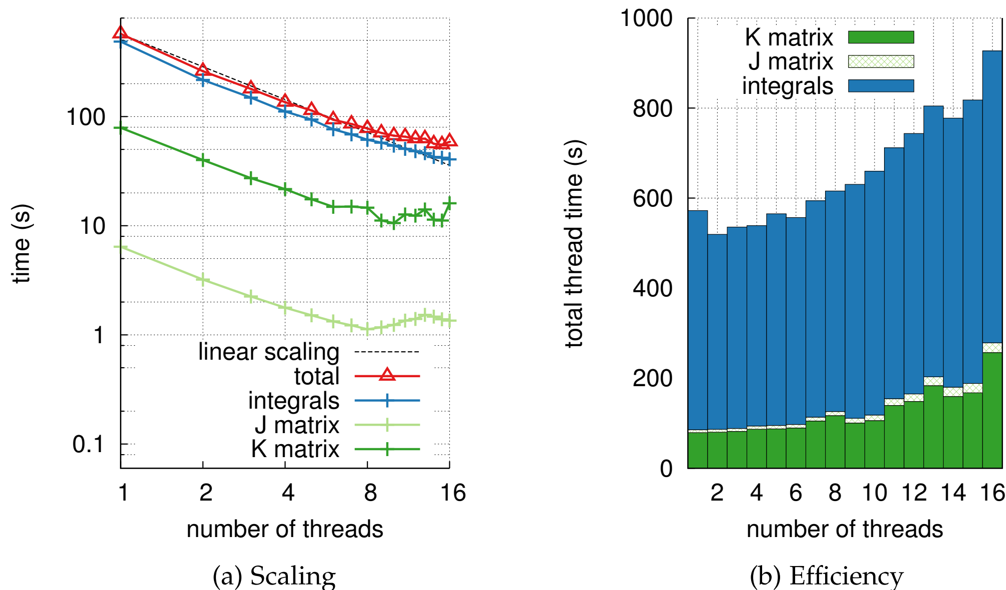


Figure 4.11: Multithreaded component scaling and efficiency of RO long range energy calculation on *Raijin* with recursive bisection of loops for integral and J matrix calculation (1–16 threads, 3D-alanine$_4$, cc-pVQZ, $\mathcal{N}' = 11, \mathcal{L} = 19$).

approach. Given the superior performance of the recursive bisection approach, we use it in all subsequent measurements.

4.2.3 Distributed-Memory Scaling

Figure 4.12 shows the strong scaling with number of X10 places on *Raijin* of Fock matrix construction, for 3D-alanine$_4$ with the cc-pVQZ basis set. Total Fock matrix construction time drops from 86.5 s on a single place (one socket) to 6.9 s on 64 places. Auxiliary integral calculation decreases with increasing number of places. K matrix calculation time decreases from 1 to 32 places, but increases above 64 places. Total computation time increases above 128 places.

Table 4.3 shows both Fock matrix and auxiliary integral load imbalance with varying numbers of places for Fock matrix construction using RO for 3D-alanine$_4$. Both measures of load imbalance tend to rise with the number of places. Auxiliary integral imbalance L_{aux} is larger than Fock matrix imbalance L_{Fock} for all numbers of places. Imbalance for 32 places is already quite severe, and limits the maximum parallel efficiency achievable to 72% assuming no parallel overhead or sequential bottlenecks. This suggests that the data distribution and load balancing schemes described in §4.1.3–§4.1.4 cannot balance load sufficiently evenly to achieve strong scaling.

As well as load imbalance, communication overhead represents a barrier to strong scaling. To compute the local portion of the K matrix, each place performs a number

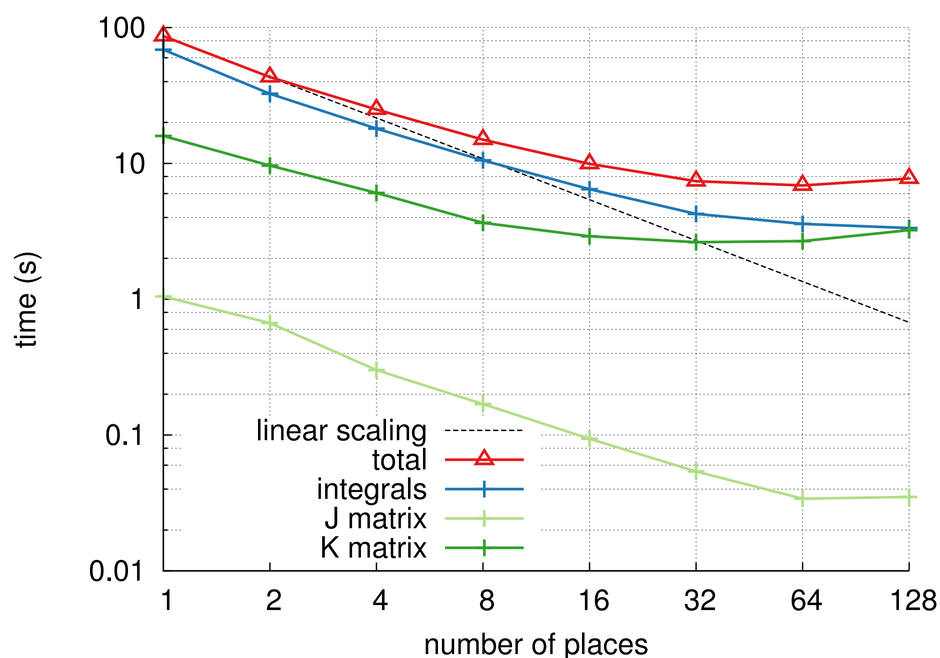


Figure 4.12: Multi-place component scaling of RO long range energy calculation (8 threads per place, 3D-alanine$_4$, cc-pVQZ, $\mathcal{N}' = 11, \mathcal{L} = 19$)

Table 4.3: Distributed Fock matrix construction using RO: calculated load imbalance between different numbers of places on *Raijin* (3D-alanine$_4$, cc-pVQZ, $\mathcal{N}' = 11, \mathcal{L} = 19$))

	Load imbalance	
places	L_{Fock}	L_{aux}
2	1.000	1.002
3	1.002	1.098
4	1.014	1.055
5	1.007	1.064
6	1.009	1.163
8	1.032	1.166
12	1.051	1.174
16	1.069	1.287
32	1.106	1.379
64	1.298	1.530

of DGEMM operations to multiply different sized blocks of B as described in §4.1.5. For each DGEMM, an $m \times k$ block of matrix B is transferred from another place and multiplied (transposed) with the local $n \times k$ block of B; the resulting $m \times n$ block is accumulated to matrix K. Thus each DGEMM requires the transfer of mk word-length matrix elements and performs $2(mnk)$ floating-point operations. Table 4.4 shows the floating-point and communication intensity for DGEMM operations in K matrix construction for the same problem on different numbers of X10 places. For each number of places, the table shows: the calculated range of floating-point intensities (in FLOPs to words), in other words the ratio of the number of FLOPs performed by DGEMM to the number of words required to be transferred between places; the measured range of floating-point performance of DGEMM in GFLOP/s; and the measured range of transfer rates in Gword/s. Theoretical peak FLOP/s on an eight-core Sandy Bridge socket of *Raijin* is 166 GFLOP/s, and MPI bandwidth is approximately 0.75 Gword/s. Therefore to achieve peak FLOP/s on *Raijin* requires a floating-point intensity of more than 220 FLOP/word.

Table 4.4: Distributed K matrix construction using RO: DGEMM floating-point and communication intensity with different numbers of places on *Raijin* (3D-alanine$_4$, cc-pVQZ, $\mathcal{N}' = 11, \mathcal{L} = 19$)

	calculated	measured	
places	**Floating-point intensity (FLOP/word)**	**Computation rate (GFLOP/s)**	**Transfer rate (Gword/s)**
2	2170 – 2170	102.4–150.4	0.041–0.072
4	1070 – 1100	110.2–145.4	0.051–0.138
8	530 – 548	90.4–134.6	0.070–0.342
16	260 – 286	48.7–106.6	0.073–0.665
32	110 – 150	12.1– 84.1	0.044–1.253
64	52 – 88	2.4– 68.7	0.017–1.206

The FLOP/word ratio drops approximately linearly with number of places, as the average size of blocks of B drops linearly with the number of places. The measured performance of DGEMM and minimum transfer rate also drop with shrinking block size. The decreasing floating-point intensity means that strong scaling hits a limit for this problem size at 32 places; after this, no further reduction in K matrix computation time is possible.

As the X10 version of RO is the only distributed implementation of the algorithm, it was not possible to perform a comparison against a reference implementation. Comparison with high-performance distributed codes for Hartree–Fock computation such as NWChem [Valiev et al., 2010] would be of benefit in future work.

4.3 Summary

This chapter described the use of the X10 programming language to implement Hartree–Fock electronic structure calculations. A distributed implementation of the linear scaling resolution of the Coulomb operator method was presented. This method has lower asymptotic scaling than conventional algorithms for Hartree–Fock calculations, and may be effectively parallelized both to reduce computation time (strong scaling) and allow the treatment of larger problems (weak scaling). The computation of Fock matrix elements is an irregular problem; X10's work stealing runtime provides load balancing between worker threads within a place, while data decomposition and transfer between places is supported by distributed data structures using X10's explicit representation of locality. In the following chapter, we consider another application with more complex distributed data structures.

Chapter 5

Molecular Dynamics Simulation Using X10

This chapter describes the application of the APGAS programming model in molecular dynamics simulation, with particular focus on the problem of calculating non-bonded electrostatic interactions. Three different methods for calculating electrostatics are implemented: direct calculation of pairwise interactions (§5.1), the particle mesh Ewald method (§5.2) and the fast multipole method (§5.3). For each method, the corresponding section first presents those features of the X10 programming language and the APGAS programming model which enable the concise expression of parallelism, data locality and synchronization within the algorithm. The improvements to the X10 programming language that were presented in chapter 3 are demonstrated in the context of the implementation. The performance of each implementation is evaluated on representative parallel computing architectures and compared with state-of-the art implementations in GROMACS [Hess et al., 2008] and exaFMM [Yokota, 2013].

Portions of this chapter have been previously published in Milthorpe et al. [2011], Milthorpe and Rendell [2012] and Milthorpe et al. [2013]. All source code included in this chapter as well as the benchmarks used in evaluation are distributed as part of ANUChem [ANUChem].

5.1 Direct Calculation

In a system of atoms interacting under non-bonded electrostatic forces, each atom experiences a force which is the sum of forces due to every other atom in the system. Likewise, the electrostatic potential of the system is the sum of potentials between all pairs of atoms in the system. The most straightforward way of evaluating electrostatic interactions is to directly calculate the force and potential between individual pairs of particles. This method may be used to calculate interactions between all pairs of particles in the system, however, as there are $\Theta(N^2)$ pairs of particles, this method is expensive for large systems. More commonly, direct calculation is only used for

```
1  for (atomI in atoms) {
2      for (atomJ in atoms) {
3          if (atomI != atomJ) {
4              val rVec = atomJ.centre - atomI.centre;
5              val invR2 = 1.0 / rVec.lengthSquared();
6              val invR = Math.sqrt(invR2);
7              val e = atomI.charge * atomJ.charge * invR;
8              potentialEnergy += 2.0 * e;
9              atomI.force += e * invR2 * rVec;
10         }
11     }
12 }
```

Figure 5.1: X10 code for direct calculation of electrostatic interactions between particles

short-range interactions up to some cutoff distance, and longer-range interactions are either neglected entirely, or else evaluated using an approximate method such as a particle mesh method (§5.2) or fast multipole method (§5.3). In any case, a large number of interactions must be evaluated directly, and an efficient method of direct calculation is required.

5.1.1 Implementation

The force and potential for a given particle are calculated by summing over its interactions with all other particles in the system. Figure 5.1 shows sequential X10 code to perform such a direct evaluation.

While the code in figure 5.1 is correct and easy to understand, it may not achieve high floating-point performance on typical computing architectures. Some of the issues that limit the performance of direct calculation, and techniques that may be used to overcome them, include:

- The division and square root operations are expensive (taking dozens of cycles) and may not be pipelined. On a typical x86 architecture such as Sandy Bridge, the combined latency of the divide and square root operations is around 40 cycles [Intel, 2011].

  Many molecular dynamics codes use a combined divide–square root function (a reciprocal square root) to reduce this cost. The GROMACS [Hess et al., 2008] generic non-bonded kernel for Coulomb interactions (nb_kernel100.c) contains 27 floating point operation (FLOP)s per interaction, including a reciprocal square root.

  Some instruction set architectures provide an approximate reciprocal square root operation, which has a significantly lower latency and may also be pipelined. Using such an operation may dramatically reduce the number of cycles required to compute a single interaction, at the cost of reduced accuracy. For example,

exaFMM [Yokota, 2013] uses single-precision Intel reciprocal square root operations which have a latency of 6 cycles. Using an approximate reciprocal square root operation, Chandramowlishwaran et al. [2010] give a figure of 19 instructions for potential evaluation only, which takes roughly 17 cycles to execute.

- To take full advantage of the floating-point capabilities of a modern CPU requires the use of vector instructions to compute arithmetic instructions for multiple interactions at once. This requires either that the programmer write vectorized code for the target architecture, or that the compiler can automatically vectorize the code. Compiler vectorization may fail if, for example, the compiler cannot determine that potential vector operands are independent.

  exaFMM [Yokota, 2013] uses hand-written AVX and SSE instructions, including a vectorized approximate reciprocal square root operation, to maximize performance on Intel architectures.

- Running both the outer and the inner loop of figure 5.1 over all N particles performs some redundant calculation. According to Newton's third law (mutual interaction), for a given pair of particles A and B, the force on A due to B may be calculated and applied to A, and then an equal and opposite force applied to B merely by reversing the sign on the force vector. Thus the number of floating-point calculations required may be roughly halved.

  However, this apparent efficiency creates problems for multithreaded computation. If mutual interaction is used, then multiple threads must update the force on each particle, which requires that the updates be synchronized. For this reason, the forces on each particle are usually computed independently without the use of mutual interaction.

In molecular dynamics simulation on distributed memory architectures, direct calculation requires the communication of updated particle positions between all processes at each timestep, which makes this method fundamentally non-scalable [Heffelfinger, 2000]. For this reason, direct calculation is typically used only for short-range interactions up to a given cutoff distance, in combination with an approximate method for treating long-range interactions.

5.1.2 Evaluation

As direct calculation of electrostatics is an important building block for larger molecular dynamics codes, it is necessary first to show that it can be efficiently implemented in X10 before considering more complex methods. The X10 code in figure 5.1 was used to evaluate potential and forces for systems of varying numbers of atoms. The number of cycles and FLOPs per evaluation was measured using PAPI [Browne et al., 2000], and compared with an equivalent C++ code. Table 5.1 shows the cycles per interaction measured for each code for various system sizes.

To evaluate a single interaction (energy and forces) using the code as written in figure 5.1 requires 9 adds, 9 multiplies, 1 division and 1 square root, for a total

Table 5.1: Direct calculation of electrostatic force: cycles per interaction, X10 vs. C++ on Core i7-2600 (one thread) for varying numbers of particles.

	cycles	
n	**X10**	**C++**
10,000	40.4	40.4
20,000	40.4	40.4
50,000	40.5	40.4
100,000	42.5	40.4

of 20 floating-point operations. There are also 10 memory operations (7 loads and three stores) on 8-byte floating-point values. The balance between floating-point and memory operations is therefore 0.25 FLOPs per byte. For both codes, a single interaction took slightly over 40 cycles to compute. This is a poor floating-point intensity of around 0.5 FLOP/cycle, largely due to the expensive square root and division operations which cannot be vectorized. The X10 code was slightly slower for larger numbers of particles, probably due to the effects of differing memory layouts on cache usage. The measured cycles per interaction are very similar between the two codes, therefore the X10 language implementation does not add significant overhead for this kernel. A parallel implementation of this kernel will be evaluated later in the context of the fast multipole method in §5.3.

5.2 Particle Mesh Ewald Method

The particle mesh Ewald method (PME) (as described in §2.6.2) avoids the need for calculating interactions between every pair of particles by splitting the force into a short-range component which falls off rapidly and a long-range component which is calculated approximately in reciprocal space. The short-range component is evaluated directly up to a chosen cutoff distance; the greater this distance, the higher the accuracy of the result. This substantially reduces the work required to calculate electrostatic interactions for large periodic molecular systems. However, computing in reciprocal space requires the use of FFTs, which implies an all-to-all communication pattern that may limit scalability.

5.2.1 Implementation

An implementation of PME was constructed in the X10 language using the improvements previously described in chapter 3. For a distributed PME, four issues merit special consideration: the domain decomposition of particle data and the charge grid between X10 places; efficient interpolation of charges to the grid; the exchange of particle data between places in calculating direct interactions; and the implementation of a distributed FFT.

5.2.1.1 Domain Decomposition With Distributed Arrays

The central data structure in PME is the charge grid array. This is a three-dimensional array of grid points of dimension $K_{[x,y,z]}$. The charge grid array is divided between places in both the x and y dimensions. This is accomplished in X10 using the two dimensional block distribution class `BlockBlockDist`.

```
1  val gridRegion = Region.make(0..Kx, 0..Ky, 0..Kz);
2  val gridDist = Dist.makeBlockBlock(gridRegion, 0, 1);
3  Q = DistArray.make[Complex](gridDist);
```

Figure 5.2: X10 code to distribute charge grid array in PME

Figure 5.2 shows the X10 code for constructing the charge grid array. Line 2 defines a block distribution over all grid points, divided first along dimension 0 (the x dimension) and then along dimension 1 (the y dimension). The division is performed so that blocks are as near as possible to square in shape to maximize data locality.

5.2.1.2 Charge Interpolation Over Local Grid Points

Long-range interactions are calculated over a gridded charge distribution [Essmann et al., 1995]. Each charge is spread over a number of grid points using B-spline interpolation. The charge spreading is challenging to parallelize due to the fact that each charge contributes to multiple overlapping grid points. If the 'owner-computes' rule is used with distributed processes, some charges must be replicated to allow each process to compute its portion of the grid. If the 'owner-stores' rule is used, synchronization is required to ensure the correct accumulation of charges to each grid point. The owner-computes rule is preferable for multithreaded computation, as it minimizes synchronization.

Our code uses lattice-centric (owner-computes) charge interpolation as suggested by Ganesan et al. [2011], but differs from their scheme in that it does not use neighbor lists. Instead, the code divides the charges into subcells with a side length equal to half the direct interaction cutoff distance. Each place considers charges in a region of subcells surrounding its resident lattice points. The same subcells are also used in the calculation of direct particle-particle interactions (see §5.2.1.3). The basic structure of the charge interpolation code is shown in figure 5.3. Line 2 uses the `getLocalPortion()` method described in chapter 3 to return the local portion of the distributed charge array `Q` as a zero-based rectangular array, allowing it to be efficiently indexed in the tight loop on lines 10–12. In contrast, if `Q` were to be used directly, this would require a virtual function lookup for each read and write access within the tight loop.

5.2.1.3 Use of Ghost Region Updates to Exchange Particle Data

For efficient implementation the particles are divided into subcells for which the side length is some rational fraction of the cutoff distance. Thus the particles in a

```
1  finish ateach(place in Q.dist.places()) {
2      val qLocal = Q.getLocalPortion();
3      val gridRegionLocal = qLocal.region;
4      val haloRegionLocal
5          = getSubcellHaloRegion(gridRegionLocal);
6      for ([x,y,z] in haloRegionLocal) {
7          for (atom in subcell(x,y,z)) {
8              // fill splines for atom
9              // accumulate splines to local grid
10             for ([k1,k2,k3] in spreadRegion) {
11                 qLocal(k1,k2,k3) += atomContribution;
12             }
13         }
14     }
15 }
```

Figure 5.3: X10 code to interpolate charges to grid points in PME

subcell interact with a defined neighborhood of subcells. Figure 5.4 shows one such neighborhood or *halo* region.

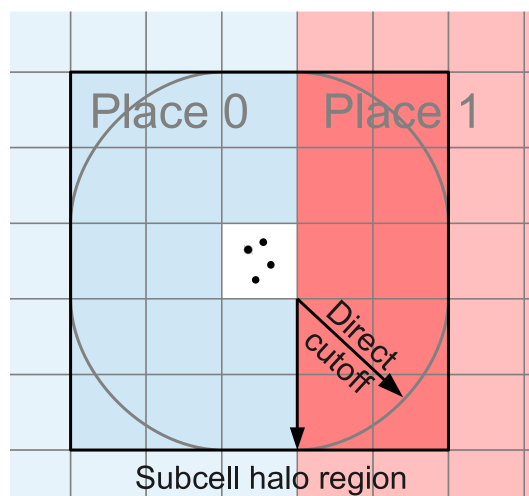


Figure 5.4: Subcell halo region used to calculate direct interaction in PME

The white square containing particles in the center of the figure represents the target subcell containing particles for which direct interactions must be calculated; the halo region comprises all subcells within the large black square, and includes all cells which are partially or wholly within the direct cutoff distance from any point in the target subcell. Note that the direct cutoff region for the white box is not a sphere but a cube with rounded edges.

Figure 5.4 also illustrates the requirement for exchange of particle data between places. The target subcell is held at place 0, along with all subcells colored blue; the

subcells colored red are held at place 1. In order to calculate direct interactions, it is necessary to communicate particles from all remotely held subcells to place 0.

An early version of the code used an application-specific ghost region update to gather subcells from neighboring places. The code was altered to use ghost region updates as described in §3.3.2, which reduced the size of the application code from 682 to 617 non-comment source lines while maintaining equivalent performance. Assuming that code size (in number of lines) corresponds to programmer effort, this example illustrates the benefit to programmer productivity of efficient high-level operations on distributed data structures.

5.2.1.4 Distributed Fast Fourier Transform

At the heart of the PME method are two 3D FFTs performed on the charge mesh. A 3D FFT may be divided between distributed places in one dimension (slab decomposition) or in two dimensions (pencil decomposition). A slab decomposition reduces the number of transpose communications required, whereas a pencil decomposition allows the grid to be divided between a larger number of places.

In our implementation, 3D FFT is decomposed into a series of 1D FFTs, interspersed with transpose operations to redistribute the data so that each place's data are complete in one dimension for each FFT [Eleftheriou et al., 2003]. The particle mesh is divided between places in both the x and y dimensions, meaning each place holds a thick pencil extending through the entire z dimension. Each of the 1D FFTs is computed by a native call to FFTW [Frigo and Johnson, 2005]. The transpose is an all-to-all communication, with the following code executed at each place:

```
1  def transpose(
2   source : DistArray[Complex](3),
3   target : DistArray[Complex](3)) {
4    finish {
5      for (p2 in source.dist.places()) {
6        val transferRegion : Region(3) = //
7        val toTransfer = // ...
8        at(p2) async {
9          var i : Int = 0;
10         for ([x,y,z] in transferRegion) {
11             // transpose dimensions
12             target(z,x,y) = toTransfer(i++);
13         }
14       }
15     }
16   }
```

5.2.2 Evaluation

To evaluate our PME implementation, we compare its performance with that of the GROMACS [1] software package, previously described in §2.6.2. GROMACS version

[1] http://www.gromacs.org/

4.5.5 was used for comparison. We evaluate first base single-threaded performance and then distributed scaling across multiple nodes of a distributed memory cluster.

5.2.2.1 Single-Threaded Performance

We performed a sequence of 1000 force calculations with the X10 PME implementation and measured the average time for the mesh component of the calculation, that is, the long range force only. As a comparison, we performed molecular dynamics runs for 1000 timesteps using GROMACS for various-sized cubic boxes of water molecules and measured the mean time for the PME mesh component of the calculation. Water boxes were generated using GROMACS `genbox` and equilibrated for 10,000 timesteps before measurement runs. A grid spacing of 0.9 nm and a direct cutoff of 1 nm were used for both codes for all systems.

Figure 5.5 shows the performance of our PME code against that of GROMACS on a single Sandy Bridge core (Core i7-2600) for a range of particle numbers between 12,426 and 1,067,037. (Non-round numbers of particles are due to the use of equilibrated cubic water boxes of fixed density.)

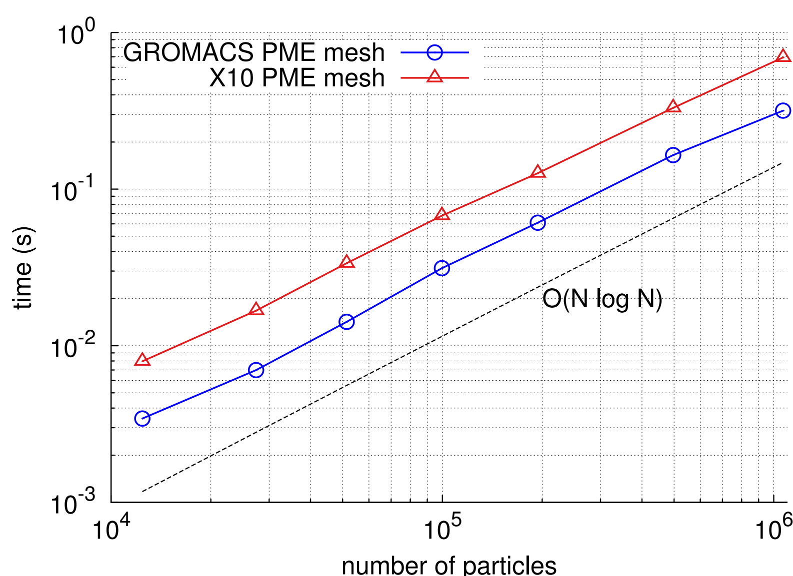


Figure 5.5: Comparison between X10 and GROMACS PME mesh evaluation on Core i7-2600 (one thread): scaling with number of particles ($\beta = 0.35$, direct cutoff = 1 nm, mesh spacing = 0.9 nm).

For the smallest system of 12,426 atoms, GROMACS takes approximately 3.4 ms and the X10 code takes approximately 8 ms per mesh evaluation. This rises in line with the expected $\mathcal{O}(N \log N)$ scaling to approximately 320 ms and 690 ms respectively for 1,067,037 atoms. Across this range of system sizes, the X10 code is approximately 2.0–2.4 times slower than GROMACS. A previously published implementation was almost two orders of magnitude slower than GROMACS [Milthorpe et al., 2011];

the difference in the current version is due largely to the efficient indexing of local data using `getLocalPortion()` as described in §5.2.1.2.[2]

5.2.2.2 Distributed-Memory Scaling

Figure 5.6 presents strong scaling results for force evaluation of a system of 33,226 water molecules in a 10 Å cubic box on *Raijin*. Also shown is a breakdown of timings for the major components of the calculation: the interpolation of charges to the mesh; FFTs; direct calculation of short-range interactions and prefetch of particle data from neighboring places.

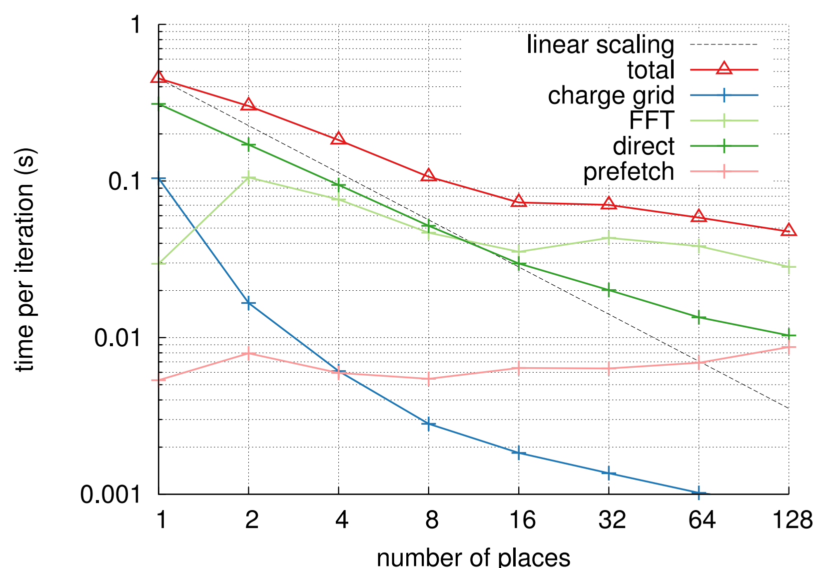


Figure 5.6: Strong scaling of PME potential calculation for 33k water molecules on *Raijin* (8 cores per place, $\beta = 0.35$, cutoff = 10 Å, grid size = 96.

The total time for evaluation on a single place (1 socket, eight cores) is 0.45 s, of which the major components are direct calculation and interpolation of charges to the grid. Total time reduces from 0.45 s on 1 place to 0.047 s on 128 places (1024 cores), due to significant reductions in both of these components. In contrast, there is no reduction in prefetch time with increasing number of places, as the number of messages per place remains roughly constant. On a single place, FFT is a small component as it may be performed as a single 3D FFT using FFTW, however, this component jumps significantly for two places as data transpose is required between places. FFT time remains roughly constant above 16 places due to the all-to-all communication required; it is therefore the limiting factor in scaling PME to large numbers of places.

[2]The cost of accessing local data through a virtual function call on a `DistArray` is paid whether or not the data are actually distributed, so the use of `getLocalPortion()` improves performance for single-place as well as multi-place execution.

5.3 Fast Multipole Method

In contrast to PME, the fast multipole method (FMM) does not require all-to-all communications and so in principle should scale better to larger numbers of nodes. In addition, compared to particle-mesh methods FMM is highly floating-point intensive, which makes it an attractive application for exascale systems [Yokota, 2013].

5.3.1 Implementation

The fast multipole method is a complex, multi-stage algorithm which allows for parallelism at multiple levels. As with PME, the most basic parallelism is between computation of near- and far-field interactions, which are independent other than the need to synchronize updates to forces on particles. Within each step of the algorithm as described in §2.6.3, evaluation of each box may proceed independently, although there are dependencies between stages and levels. For example, the local expansion of the parent box is an input to the computation of the local expansion of each child box.

Our code, called *PGAS-FMM*, uses activities of box-level granularity, and overlaps computation with communication where possible. At a high level, the fetching of particle data for near-field interactions (*P2P*) overlaps with all bar the final phase of computation, as follows:

```
1  finish ateach(place in Place.places()) {
2    finish {
3      async fetchParticleData();  // required for P2P
4      upwardPass();               // P2M, M2M
5      multipolesToLocal();        // M2L
6    }
7    downwardPass();               // L2L, L2P, P2P
8  }
```

In the above code, the **async** statement on line 3 starts an activity to fetch particle data from neighboring places. Each place constructs a local essential tree [Warren and Salmon, 1992], which contains only that portion of the tree that is necessary to compute interactions for boxes owned by the current place. Fetching particle data for boxes in the local essential tree proceeds in parallel with the `upwardPass()` and/or `multipolesToLocal()` phases. The inner **finish** block (lines 2–6) ensures that fetching is completed before the `downwardPass()`, which includes near-field computation (*P2P*).

Calculation of far-field interaction uses spherical harmonics with rotation-based translation and transformation operations as described by White and Head-Gordon [1996], with rotation matrix relations as given by Dachsel [2006].

FMM, like other tree-structured codes, operates by means of *traversals* of the tree. A *pre-order traversal* operates first on boxes at the highest level of the tree, then recursively on child boxes. In contrast, a *post-order traversal* operates first on child boxes. We divide each tree traversal into one activity for each box. Therefore the

number of activities in a traversal of a tree of $D_{\max}$ levels is

$$8^{D_{\max}} + 8^{D_{\max}-1} + ... + 8^2. \tag{5.1}$$

(The topmost level of boxes is not included in the traversal as there are no far-field evaluations at this level.) For example, for a complete tree of 5 levels, the number of activities for a traversal is 37440.

In implementing these activities for efficient computation on a distributed architecture, two significant issues are the construction of a distributed tree structure, and load balancing the activities between places in the computation. In our implementation, a third issue arises from the use of a global load balancing approach: the need for efficient global collective communication. We consider these three issues in the following sections.

5.3.1.1 Distributed Tree Structure Using Global References

The tree is composed of lowest-level boxes containing particles, represented by the class `LeafOctant`, and boxes at higher levels, represented by the class `ParentOctant`. Both classes inherit from `Octant`, defined as follows:

```
1  public abstract class Octant {
2    public id:OctantId;
3    public var parent:Octant;
4    public val multipoleExp:MultipoleExpansion;
5    public val localExp:LocalExpansion;
6    abstract traverse[T,U](parentRes:T,
7      preFunc:(a:T)=>T,
8      postFunc:(c:List[U])=>U
9    ):U;
10   ...
11 }
```

The following code performs the traversal of a subtree for a parent octant:

```
12 class ParentOctant extends Octant {
13   public val children:Rail[Octant];
14   traverse[T,U](parentRes:T,
15     preFunc:(a:T)=>T,
16     postFunc:(c:List[U])=>U
17   ):U {
18     val myRes = preFunc(parentRes);
19     val childRes = new Rail[T](numChildren);
20     finish for([i] in children) {
21       async childRes[i] =
22         children[i].traverse(myRes, preFunc, postFunc);
23     }
24     val x = postFunc(childRes);
25     // store for use by ghost
26     atomic this.result = x;
27     return result;
```

```
28    }
29    traverse[T,U](postFunc:(c:List[U])=>U):U {
30      val childRes = new Rail[T](numChildren);
31      finish for([i] in children) {
32        async childRes[i] =
33          children[i].traverse(postFunc);
34      }
35      val x = postFunc(childRes);
36      return result;
37    }
38  }
```

On lines 15 and 16, `preFunc` and `postFunc` are arbitrary *closures*. They are used in FMM to translate or transform multipole expansions. Lines 20–23 traverse each child in parallel using the result of `preFunc` for the parent (a pre-order traversal). Line 24 executes `postFunc` for the parent, which typically involves a reduction over the result of `postFunc` for all children (a post-order traversal). Line 26 atomically sets the result field for the parent octant. The **`atomic`** statement is necessary to allow progress for any activities that are currently waiting on the result field. Lines 29–37 are a simplified version of traversal for the case where only post-order traversal is required.

The PGAS programming model in X10 allows for the construction of distributed pointer-based data structures. X10 provides a special type `GlobalRef`, which is a global reference to an object at one place that may be passed to any other place. Rather than using `GlobalRef` directly for child or parent references, we use it to implement a proxy class, `GhostOctant`, which sends an active message to the host place of a remote box. If a child box is held at a different place to its parent, a `GhostOctant` is created for the child and linked from the parent box.

During tree traversal, the `GhostOctant` creates an active message to the child's host place to get the aggregate representation of the child. If a pre-order traversal is required, the active message initiates computation at the host place as follows:

```
1  class GhostOctant extends Octant {
2    private var target:GlobalRef[Octant];
3    traverse[T,U](parentRes:T,
4      preFunc:(a:T)=>T,
5      postFunc:(c:List[U])=>U
6    ):U {
7      at(target.home) {
8        val t = target();
9        return t.traverse(parentRes, preFunc, postFunc);
10     }
11   }
12   ...
```

Lines 7–10 generate an active message to the home place of the target octant to compute and return the aggregate representation of the octant.

If only post-order traversal is required, then the various subtrees at each place can be evaluated in parallel, without waiting for evaluation of parent octants held

at other places. In this case, a `GhostOctant` can simply return the value of the host octant that has previously been computed at the host place. Continuing on from the previous listing:

```
12    ...
13    traverse[U](postFunc:(c:List[U])=>U):U {
14      at(target.home) {
15        val t = target();
16        when(t.result != null);
17        return t.result;
18      }
19    }
```

The statement `when(...);` on line 16 is a conditional atomic statement, i.e. it blocks the current activity until the condition is true, during which time other activities can perform useful work. The thread may proceed once the condition has been set to true by an atomic block in another activity initiated separately at the host place.

5.3.1.2 Load Balancing

Exposing parallelism through parallel activities does not by itself ensure the efficient use of available processing resources. Load balancing between processing elements is necessary to ensure full utilization of resources.

Load balancing FMM presents two challenges: the difference in number of potential interactions for boxes on the edge of the simulation space compared with those for interior boxes, and differences in the work lists for near- and far-field interactions [Kurzak and Pettitt, 2005]. The first issue does not arise in molecular dynamics simulations with periodic boundary conditions, as with the periodic FMM every box can be considered an 'inside' box (with 26 neighbors). However for our simulation it is important because periodic boundary conditions do not apply. The second issue occurs because the major components of the near- and far-field interactions — the *P2P* and *M2L* steps described in §2.6.3 — are proportional to the number of particles within neighboring boxes and the number of boxes in well-separated boxes respectively. As these values are not directly correlated, the balance of work for each component will be different for each box. One approach to load balancing uses a separate domain decomposition for the near- and far-field computations [Kurzak and Pettitt, 2005]. An alternative is to use a single domain decomposition based on a combined work estimate for both components [Lashuk et al., 2009].

Load Balancing Between Places

We chose to implement a single domain decomposition of the FMM tree based on work estimates for each box at the lowest level. A full work estimate for FMM would contain many terms, accounting for computation and communication in each stage of the algorithm. However, a simple two-part estimate is possible based on two simplifying assumptions: first, that FMM is heavily compute-bound on current architectures (although this assumption may become false as early as 2020) [Chandramowlishwaran

et al., 2012]; and second, that only the direct interaction (particle-to-particle or *P2P*) and multipole-to-local (*M2L*) steps contribute significantly to the total runtime. Given these assumptions, it is possible to estimate the work due to a box B containing n particles, given an average q particles per lowest level box.

The cost of computing a single direct interaction C_{P2P} and the cost of computing a single multipole-to-local transformation C_{M2L} may be estimated at runtime as part of the 'setup' for the method by executing a small number of *P2P* and *M2L* kernels on the target architecture. The U-list $U(B)$ is the list of all neighboring boxes. *P2P* (particle-to-particle) interactions are computed for all n particles in box B with every particle in all boxes in the U-list. Therefore the cost of computing near-field interactions for box B is

$$\text{cost}_u(B) = C_{\text{P2P}} \cdot n \cdot |U(B)| \cdot q. \tag{5.2}$$

The V-list $V(B)$ is the list of all well-separated boxes for which the parent boxes are not also well separated. *M2L* transformations are computed for all boxes in the V-list. Therefore the cost of computing far-field interactions for box B is

$$\text{cost}_v(B) = C_{\text{M2L}} \cdot |V(B)|. \tag{5.3}$$

The estimate for the total work due to box B is simply

$$\text{cost}(B) = \text{cost}_u(B) + \text{cost}_v(B). \tag{5.4}$$

The cost estimate above is used to divide the lowest-level boxes between places. Boxes are sorted using Morton ordering and portions of the Morton curve are assigned to each place so that the cost of each portion is roughly equal.

Load Balancing Within a Place

The work required for traversal varies substantially between boxes; it is therefore necessary to load balance activities between the worker threads within a place. This is performed by the X10 work stealing runtime; once a thread becomes idle it attempts to steal work from the queue of another thread. Work stealing leads to good locality when applied to tree traversals, as the earliest created (and stolen) activities are those at the higher levels of the tree. The lower level activities created and processed by each thread will therefore tend to belong to the same subtree, which means that they can reuse cached data pertaining to that subtree. As boxes are stored in memory in Morton order, boxes that are close together in space also tend to be close together in memory. The locality generated by work stealing within an FMM force calculation will be evaluated in §5.3.2.3.

5.3.1.3 Global Collective Operations

For an algorithm to be truly scalable it cannot include global synchronous communications, as was seen previously with the all-to-all communications limiting the scalability of PME in §5.2. However, if such communications make up only a small portion of the runtime for small numbers of places, they may not prevent scaling the code to problem sizes of interest. Furthermore, an algorithm which uses global properties or global synchronization may be significantly simpler to implement than one which only relies on local or asynchronous communication.

The load-balancing approach described in §5.3.1.2 is an example of such an algorithm. Each place determines how many boxes should be transferred to or from neighboring places by comparing the cost estimate for the boxes it currently holds with the total cost estimate for the system. To determine the total cost estimate, however, requires the local costs at each place to be combined using a global all-reduce operation over all places.

During far-field evaluation, a global barrier is used to ensure that all multipole expansions have been transferred before commencing the multipole-to-local (M2L) transformations. Barrier, all-reduce and other collective operations are provided by the `x10.util.Team` API, and are hardware accelerated where possible [Grove et al., 2011]. Exploratory benchmarks of these collective operations suggested that they are not a limiting factor for the small to moderate numbers of places considered here. For very large place counts this is likely, however, to become a bottleneck. At that point one alternative approach would be to communicate the multipole and particle data using active messages with only local synchronization between neighbors [Milthorpe and Rendell, 2012].

5.3.2 Evaluation

In evaluating the performance of PGAS-FMM, we compare it to a state-of-the-art implementation, and determine whether performance differences are due to algorithm or technology. Specifically we consider i) the overall performance of the code when running on a single core and the efficiency of each major component of the algorithm; ii) the overhead of X10 activity management; iii) multithreaded scaling of each component; and iv) distributed scaling on two multiple places for typical HPC architectures with different CPU and network characteristics.

All simulations reported in this section used uniform lattice distributions over $[-1,1]^3$ with total charge $\sum q_i = 1$ and equal particle charges of $\frac{1}{n}$. As FMM permits a tradeoff between accuracy and computation time, we compare two different settings for the parameter p, the number of terms in the multipole and local expansions. These are: low-accuracy calculation with $p = 3$, which for the systems considered here corresponds to a root-mean-squared force error of $\epsilon \leq 10^{-2}$ (potential error $\epsilon_U \leq 10^{-3}$); and a high-accuracy calculation with $p = 6$, which corresponds to a root-mean-squared force error $\epsilon \leq 10^{-4}$ (potential error $\epsilon_U \leq 10^{-6}$). While the code supports changing the value of the well-spacedness parameter ws, all simulations reported here use a well-spacedness of 1, which is the only value supported by the

comparison implementation.

5.3.2.1 Single-Threaded Performance

We first compare the single-threaded performance of our code against exaFMM [exaFMM]. We used both codes to calculate forces and potential to low accuracy ($p = 3$, RMS force error $\epsilon \leq 10^{-2}$) for a range of particle numbers between 10,000 and 1,000,000. Figure 5.7 compares the performance of PGAS-FMM with that of exaFMM on a single Sandy Bridge core (Core i7-2600). For low accuracy calculations[3], our code is between 9 and 25 times slower than exaFMM across this range. This difference is similar to that observed by Yokota [2013] when the performance of exaFMM was compared with four other major open-source FMM or tree code implementations for a similar benchmark.

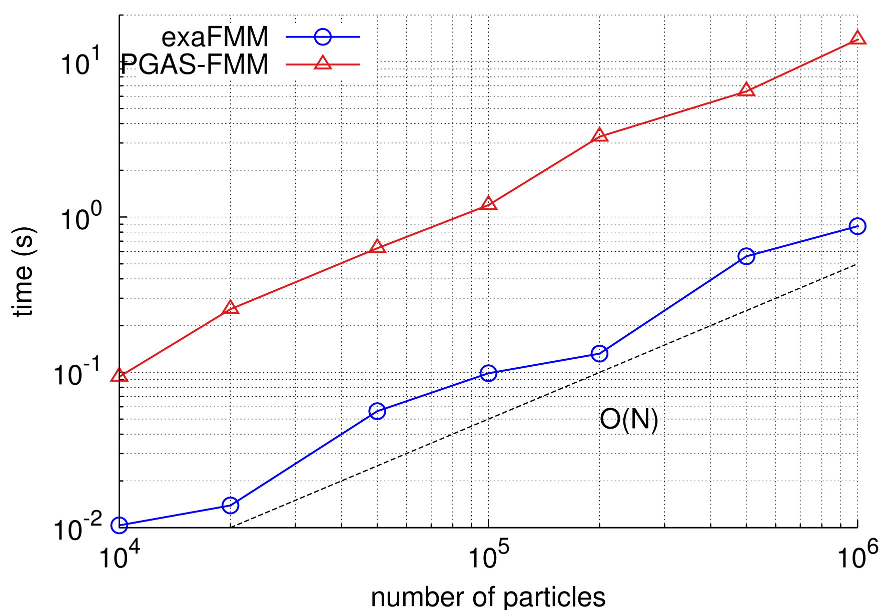


Figure 5.7: Low accuracy ($p = 3, \epsilon = 10^{-2}$) comparison between PGAS-FMM and exaFMM on Core i7-2600 (one thread): scaling with number of particles.

We now focus on the execution time of PGAS-FMM, and consider the major components of the algorithm separately. Table 5.2 shows the breakdown of single-threaded computation time between the major components: tree construction, far-field evaluation (upward, multipole-to-local transformations and downward pass) and near-field evaluation.

The overall scaling is roughly linear in the number of particles, as expected. Tree construction accounts for approximately 4-10% of the total time; this is somewhat slower than what might be expected for a single thread, in part because the code

[3] Timings for high accuracy calculations will be reported later in this section.

Table 5.2: Component timings in seconds of PGAS-FMM on Core i7-2600 (one thread) for varying numbers of particles and accuracies (low: $p = 3, \epsilon = 10^{-2}$, high $p = 6, \epsilon = 10^{-4}$).

		$n = 10^5$, $D_{\max} = 4$		$n = 10^6$, $D_{\max} = 5$	
	component	**low accuracy**	**high accuracy**	**low accuracy**	**high accuracy**
	tree construction	0.038	0.038	0.34	0.35
	near	0.714	0.718	9.37	9.28
far	upward	0.118	0.150	1.15	1.44
far	M2L	0.286	0.775	2.75	7.31
far	downward	0.052	0.097	0.52	0.96
	total	**1.21**	**1.78**	**14.13**	**19.35**

assumes that the particles are distributed between places and require load balancing and redistribution before evaluation can occur. For low accuracy the near-field computation time is dominant representing 58% of the total for the "small" 10^5 particles simulation and 66% for the "large" 10^6 particles simulation. However, as the accuracy is increased the near and far field computation times become roughly equal.

As X10 is a new language with less well-understood performance characteristics than traditional languages such as C++ or Fortran, it is reasonable to ask how good the absolute performance of PGAS-FMM is, and what fraction of the observed performance difference between PGAS-FMM and exaFMM is due to algorithmic differences compared to inefficiencies in the language implementation. These issues are considered in detail in the following subsection for the near- and far-field components separately.

Near-Field Calculation

The U-list (near-field) interactions for each box are computed in a loop over neighboring boxes. All particle data for neighboring boxes have previously been retrieved and stored in Morton order, i.e. in a dense memory layout; this promotes better locality and cache re-use. For each box, the maximum number of boxes in the U-list is 27. There is a substantial overlap between U-lists for different boxes that are nearby in Morton order. Hence there is good temporal locality in the use of particle data in the U-list calculation. Given an average number of ions $q = 50$ per lowest level box, the working set size is approximately 42 KiB, which easily fits within the L2 cache on all the computing systems we used.

As previously discussed in §5.1.2, evaluation of direct force and potential takes approximately 40 cycles per near-field interaction, which is the same as an equivalent C++ code. Near-field evaluations in exaFMM are significantly faster due to the use of AVX and SSE approximate reciprocal square root operations.

Far-Field Calculation

The major algorithmic consideration in the far-field calculation is the type of expansions and transformation operators used. By far the largest contribution to calculation time is the multipole-to-local (M2L) transformation, which converts a multipole expansion for a well separated box to a local expansion for a target box. The M2L operations on the Cartesian Taylor expansions used in exaFMM scale as $\mathcal{O}(p^6)$, whereas the rotation-based operators in PGAS-FMM scale as $\mathcal{O}(p^3)$ but with a larger prefactor. It is therefore expected that PGAS-FMM would exhibit relatively better performance for higher-accuracy calculations with greater p.

Yokota [2013] (fig. 1) compared the performance of M2L operator using hand-optimized Cartesian Taylor expansions and transformations with a less carefully optimized rotation-based M2L operator over spherical harmonic expansions. The average time for a single M2L transformation was measured for a complete FMM calculation for $n = 10^6$ particles. Yokota found that transformations using Cartesian Taylor expansions were faster for $p < 12$, whereas spherical harmonics were faster for greater values of p.

We replicated this experiment using the spherical harmonic expansions and operators implemented in PGAS-FMM, and report the results in figure 5.8. Whereas Yokota found a crossover point at $p \approx 12$ (which corresponds to a RMS force error $\epsilon \approx 10^{-7}$), we find that the crossover occurs much earlier at $p = 7$ (which corresponds to a RMS force error $\epsilon \approx 10^{-5}$).

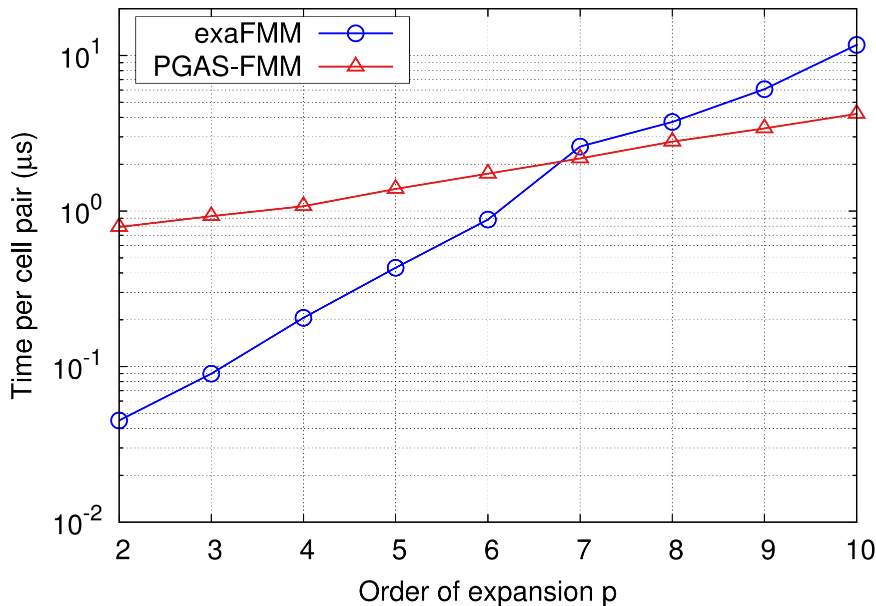


Figure 5.8: Time for M2L transformation on Core i7-2600 (8 threads) for different orders of expansion p ($n = 10^6$)

Figure 5.9 shows computation time for the same systems used in figure 5.7, this

time with a slightly higher accuracy (RMS force error $\epsilon \approx 1 \times 10^{-3}$). The exaFMM C++ code is now only 1.5–4.1 times faster than our X10 code ($p = 6$). The difference in performance suggests that algorithmic differences between the two codes are likely to be more important for far-field evaluation than differences due to the use of X10 versus pure C++.

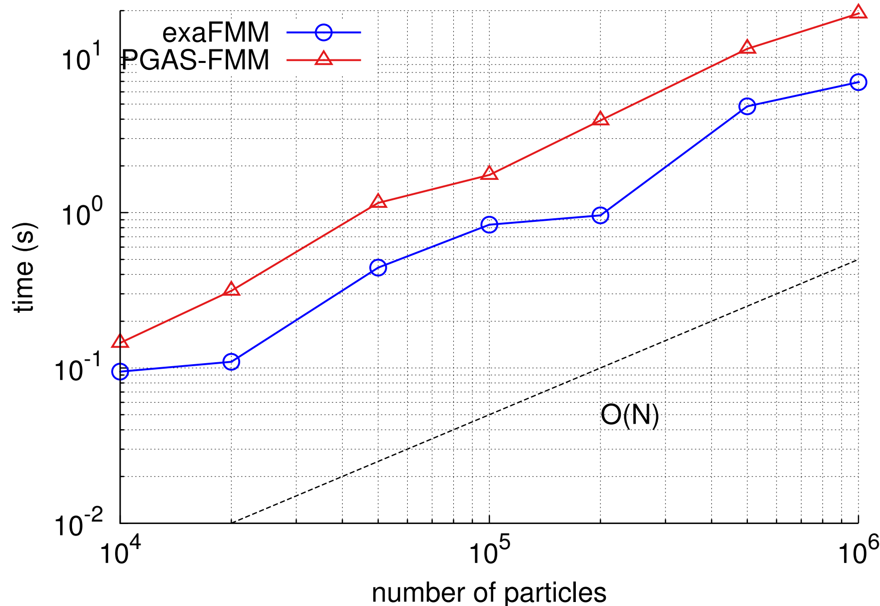


Figure 5.9: Higher accuracy ($p = 6, \epsilon = 10^{-4}$) comparison between PGAS-FMM and exaFMM on Core i7-2600 (one thread): scaling with number of particles.

Having compared the overall performance of our code to that of a highly-optimized FMM implementation, we now consider its floating-point performance with respect to theoretical peak FLOP/s on our desktop system. Our M2L operations average 3782 cycles and 3938 FP instructions with $p = 6$ (force error $\leq 10^{-4}$) on an Intel Core i7-2600. This is a FP intensity of 1.04 FLOP/cycle, which is 26% of peak FLOP/cycle. Our code achieves between 14% and 22% of peak FLOP/s for the entire FMM excluding the near-field calculation. This might be improved through appropriate use of SSE and/or AVX instructions.[4]

5.3.2.2 Overhead of Activity Management

Having considered the factors affecting sequential performance of PGAS-FMM, we next measured the overhead due to dividing the computation for parallel execution. Our code divides the work to create a single activity per box[5] to take advantage of

[4]Inspection of the generated assembler indicates that the compiler does not generate these instructions for key loops in the M2L kernel.

[5]Actually, three activities per box: one each for the upwards pass, downwards pass and multipole-to-local transformations.

dynamic load-balancing by X10's work stealing runtime. As each activity is long-lived (>100k cycles), the overhead of activity management and load balancing is expected to be small. To assess the overhead of activity management we removed all `async` statements from the upward, M2L and downward components of the code, effectively converting it into a sequential program with a single activity. Both single-activity and parallel versions of the code were run on a single thread to compare component timings. Table 5.3 shows the difference in time between single-activity and parallel versions of each component, and the proportional slowdown due to activity management overhead. The slowdown is less than 4% for each component suggesting that the sequential overhead due to activity management is low.

Table 5.3: Slowdown due to X10 activity management overhead for PGAS-FMM on Core i7-2600 (one thread) for low accuracy calculation ($n = 10^6$, $p = 3$, $\epsilon = 10^{-2}$).

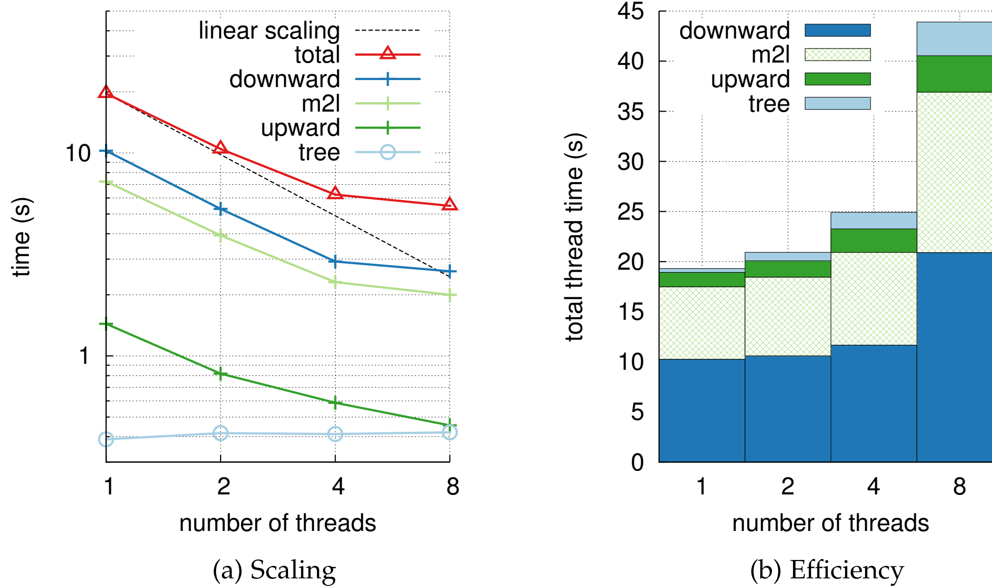
	time (s)		
component	**single-activity**	**parallel**	**slowdown**
upward	1.12	1.16	1.03
M2L	2.65	2.67	1.009
downward	9.79	9.81	1.002

5.3.2.3 Shared-Memory Scaling

We next measured the reduction in total computation time and component times for PGAS-FMM when running on different numbers of threads of a single-shared memory node. Figure 5.10(a) shows multithreaded scaling on a single quad-core Sandy Bridge node of the different components of PGAS-FMM for the largest system used in §5.3.2.1 (10^6 particles) with $p = 6$ (RMS force error $\approx 1 \times 10^{-3}$). Figure 5.10(b) presents the same data in terms of parallel efficiency, showing the total thread time (elapsed time $\times$ number of threads).

The total time reduces from 19.2 s on a single thread to 5.07 s on 8 threads. On a single thread, the largest components are the 'downward pass', which includes near-field interactions, and the multipole-to-local transformations for the V-list. For these components, figure 5.10(a) shows a significant reduction in computation time from 1 to 4 threads, while figure 5.10(b) shows a slight increase in total thread time reflecting imperfect load balancing due to differences in task size. There is a further reduction in computation time for 8 threads as hyper-threading is used to schedule eight threads on four physical cores. The latter does however increase the total thread time as threads compete for resources. Further experiments (not shown) found no additional performance improvement above 8 threads.

The locality of work stealing applied to activities over the FMM tree can be visualized using the approach that was presented in §3.1.2. Figure 5.11 shows the mapping from activity to worker thread for a slice through the simulation space at



(a) Scaling (b) Efficiency

Figure 5.10: Multithreaded component scaling and efficiency of PGAS-FMM on Core i7-2600 (1-8 threads, $n = 10^6$, $p = 6$, $\epsilon = 10^{-4}$).

the lowest level of a uniform tree of $D_{\text{max}} = 4$ levels. There is one activity for each box, thus this slice represents 4096 activities. Each worker processes a few contiguous regions of boxes, with the minimum extent of a region in any dimension being 4. Therefore, although work stealing permits fine-grained load balancing of activities between workers, in this application the overall effect is a coarse-grained division of the simulation space, with good locality between the activities processed by each worker.

5.3.2.4 Distributed-Memory Scaling

To evaluate distributed scaling we measured the time for PGAS-FMM force calculation for 1,000,000 particles using different numbers of nodes of *Raijin*. The maximum tree depth for this problem size is 5 (32,768 boxes at the lowest level), and $p = 6$ terms were used in expansions for a force error of approximately 10^{-4}. Strong scaling experiments were also conducted on the *Watson 2Q* Blue Gene/Q system. Blue Gene/Q represents a different system balance to the *Raijin* Sandy Bridge/IB cluster, with a greater relative performance of the communication subsystem compared to floating-point computation [Haring et al., 2012]. It also has substantially greater levels of parallelism; a single BG/Q compute node may execute up to 64 hardware threads (on 16 4-way SMP cores).

Figure 5.12 shows the strong scaling measured on *Raijin*.

Total computation time is shown along with the time for each of the major components for 1 to 128 places. Total time reduces from 3.6 s on a single place (8 cores)

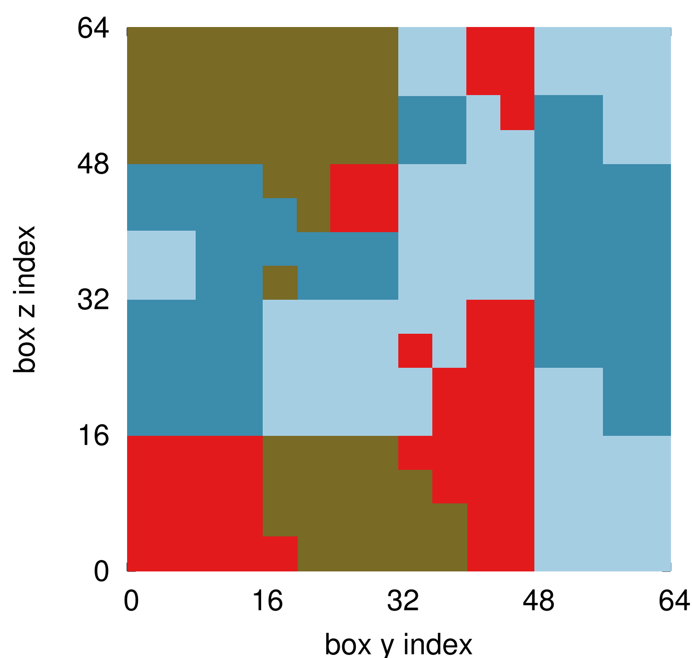


Figure 5.11: Locality of activity-worker mapping for FMM force evaluation on Core i7-2600 (leaf boxes at $x = 3$, $n = 10^6$, $D_{\max} = 6$, `X10_NTHREADS=4`). Activities executed by each worker thread are shown in a different color.

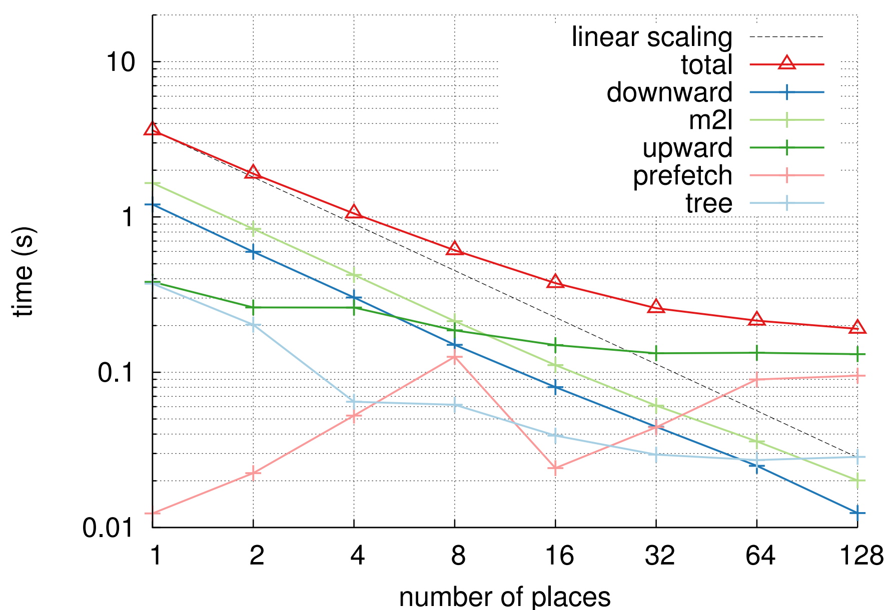


Figure 5.12: Strong scaling of FMM force calculation on *Raijin* (8 cores per place, $n = 10^6$, $p = 6$).

to 0.19 s on 128 places (1024 cores). Parallel efficiency reduces gradually due to poor scaling of the upward pass, which includes the time to send multipole expansions to neighboring places. The upward pass includes communication and synchronization between each place and its neighbors. Figure 5.12 shows an additional component, which is the time to prefetch particle data required for near-field interactions at each place. This is insignificant below 32 places but increases to become the second-largest component of the runtime on 128 places.

Figure 5.13 shows the strong scaling measured on *Watson 2Q*.

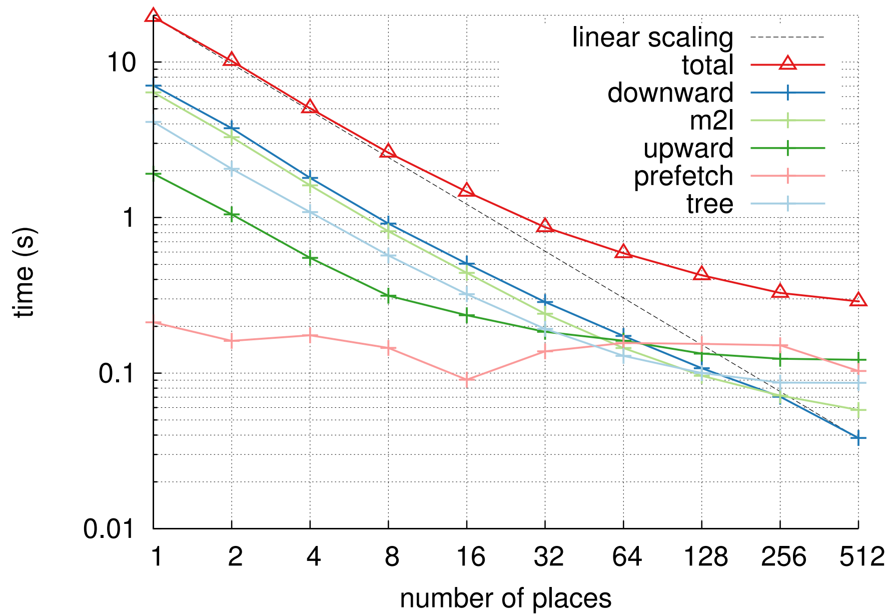
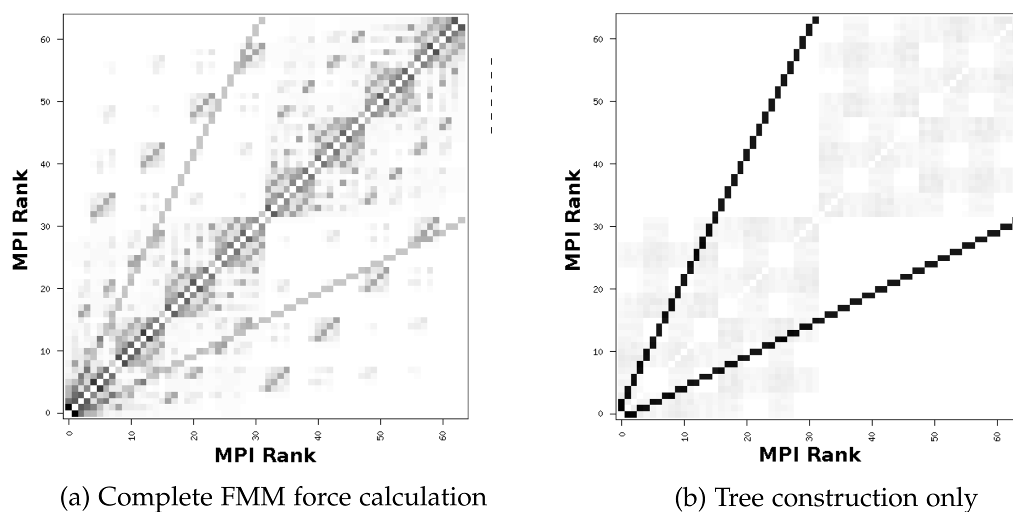


Figure 5.13: Strong scaling of FMM force calculation on *Watson 2Q* (16 cores per place, $n = 10^6$, $p = 6$).

For one place (16 cores) *Watson 2Q* takes 19.5 s in total, which reduces to 0.28 s on 512 places (8192 cores). The time for a single place (16 cores) is about 4.8 times as long as a single place on *Raijin* (8 cores). Per core, Watson is therefore more than 9 times slower than *Raijin*. The computation overall scales better on *Watson 2Q* than it does on *Raijin*, which reflects the relatively higher performance of communications over the torus network, which makes communication-intensive components like the upward pass relatively cheaper on BG/Q. Also, key collective operations used in tree construction and the upward pass (see §5.3.1.3) are hardware accelerated.

As previously mentioned, an attractive feature of FMM in comparison to particle-mesh methods is that it uses localized as opposed to all-to-all communication patterns. Figure 5.14(a) is a heatmap of the MPI pairwise communications with 64 processes on *Raijin* for a single FMM force calculation, profiled using IPM. Darker areas on the map indicate larger volumes of communications between processes. The heatmap demonstrates locality in the communication pattern, with large amounts of data exchanged between neighboring processes and very little between more distant

processes. A noticeable feature of the communication topology is the two strong off-diagonal lines. These represent global tree-structured collective communications (broadcast and all-reduce), which are used in tree construction.[6] Figure 5.14(b) shows the communication topology for tree construction alone. Comparing the two figures, it is apparent that the communication pattern of FMM evaluation excluding tree construction is fractal-structured and mostly on-diagonal (between neighboring processes).



(a) Complete FMM force calculation (b) Tree construction only

Figure 5.14: Map of MPI communications between 64 processes for FMM force calculation on *Raijin*.

5.4 Simulating Ion Interactions in Mass Spectrometry

As a practical application of the FMM code developed in the previous section, the code was used to evaluate ion interactions in mass spectrometry. We aim to simulate the behavior of a packet of ions in Fourier transform ion cyclotron resonance mass spectrometry (FTICR-MS) (see §2.6.4) over an experimentally meaningful timescale. A typical packet contains 10^4–10^6 ions in circular motion with cycle times on the order of microseconds. Simulation of these ions involves evaluating ion-ion interactions, the influence of electric and magnetic fields on each ion, and integration of equations of motions. As ion density is very low[7], in our simulation the ions are modeled as point charges, interacting with each other only through non-bonded electrostatic forces.

For adequate frequency resolution, a measurement period of tens of milliseconds is required. However, to avoid significant error in integrating the ion trajectories requires a simulation timestep of tens of nanoseconds [Birdsall and Langdon, 1985;

[6] The MPI collective functions `MPI_Bcast` and `MPI_Allreduce` exhibit similar communication patterns.

[7] on the order of hundreds to thousands of ions per cubic millimeter

Patacchini and Hutchinson, 2009]. Thus a full simulation requires around a million timesteps. The Penning trap which contains the ions is cubic in shape, which is highly convenient for simulation in an octree of cubic boxes. As ions can hit the wall of the trap and effectively disappear, periodic boundary conditions should not be imposed if the simulation is to accurately reproduce experiment. Instead, particles that venture outside of the domain defined by the octree should simply be removed from the simulation. The non-periodic nature of the experimental environment plus the dominance of the computation time by evaluation of long-range electrostatic interactions makes FMM the ideal method to use for this problem.

5.4.1 Implementation

Ions are confined radially by a magnetic field **B** and an ideal quadrupolar electric trapping potential of V_T. The electrostatic potential and field near the center of the trap are approximated as

$$\Phi_T(x,y,z) = V_T(\gamma' - \frac{\alpha'}{2l^2}(x^2 + y^2 - 2z^2)) \tag{5.5}$$

$$\mathbf{E} = -\nabla\Phi_T(x,y,z) = \frac{\alpha'}{l^2}(-x\mathbf{i} - y\mathbf{j} + 2z\mathbf{k}), \tag{5.6}$$

where $\gamma' = 1/3$ and $\alpha' = 2.77373$ are geometric factors for the cubic trap, and l is the edge length [Guan and Marshall, 1995]. The force experienced by an ion in the trap due to the confining fields is

$$\mathbf{F} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}). \tag{5.7}$$

Charges and mass are simulated in atomic units, lengths in nm and time in ns. The force on each ion due to the confining fields is added to that due to Coulomb repulsion by other ions, calculated using the FMM code described in §5.3.

5.4.1.1 Integration Scheme

Particle positions and velocities are updated using the Boris integrator [Boris, 1970] as formulated by Birdsall and Langdon [1985]. This is a modified leapfrog scheme in which positions are calculated at times $..., n-1, n, n+1, ...$ and velocities at times $..., n^{-1/2}, n^{+1/2}, ...$ as follows:

$$\mathbf{v}^- = \mathbf{v}^{n-1/2} + \frac{q\mathbf{E}}{m}\frac{\Delta t}{2} \tag{5.8}$$

$$\mathbf{v}' = \mathbf{v}^- + \mathbf{v}^- \times \mathbf{t} \qquad \mathbf{t} = \frac{q\mathbf{B}}{m}\frac{\Delta t}{2} \tag{5.9}$$

$$\mathbf{v}^+ = \mathbf{v}^- + \mathbf{v}' \times \mathbf{s} \qquad \mathbf{s} = \frac{2\mathbf{t}}{1 + t^2} \tag{5.10}$$

$$\mathbf{v}^{n+1/2} = \mathbf{v}^+ + \frac{q\mathbf{E}}{m}\frac{\Delta t}{2} \tag{5.11}$$

5.4.1.2 Ion Motion

The cyclotron angular velocity ω_c is predicted by $\omega_c = \frac{qB}{m}$, with cyclotron frequency in S.I. units $\nu_c = \frac{\omega_c}{2\Pi}$ [Guan and Marshall, 1995]. To achieve 1% phase error in simulated cyclotron frequency requires $\omega_c \Delta t \lesssim 0.3$ [Birdsall and Langdon, 1985; Patacchini and Hutchinson, 2009].

Ions are excited to a uniform radius r by an alternating electric field. In the simulation, initial particle positions are assigned in a cylinder centred at r using uniform distributions over radius, angle and height. The magnitude of ion velocity in the xy plane (in S.I. units) is predicted by

$$v_{xy} = \frac{qBr}{m}. \tag{5.12}$$

Initial velocities are generated by adding v_{xy} to a Maxwell distribution at 300 K.

The electric field gives rise to a magnetron motion of a lower frequency ω_z. In a quadrupolar trapping potential of V_T, the modified cyclotron frequency ω_+ is predicted by

$$\omega_+ = \frac{\omega_c}{2} + \sqrt{\frac{\omega_c^2}{4} - \frac{\omega_z^2}{2}} \qquad \omega_z = \sqrt{\frac{2\alpha q V_T}{ml^2}} \tag{5.13}$$

In experiment, the modified cyclotron frequency may be measured by peaks in the Fourier transform of the induced current time signal.

5.4.1.3 Induced Current

In experiment, the current is measured between detector plates on opposite walls of the cube parallel to the magnetic field.

In the simulation, a current I is induced by the movement of the 'image' $\mathbf{E}_{\text{image}}(\mathbf{r})$ associated with each ion, that is, the difference in the electric field generated by the ion at each of the two detector plates [Guan and Marshall, 1995].

$$I = \sum_{i=1}^{N} q_i \mathbf{v}_i \cdot \mathbf{E}_{\text{image}}(\mathbf{r}_i) \tag{5.14}$$

$$\mathbf{E}_{\text{image}}(\mathbf{r}) = -\frac{\beta'}{l} r_j, \tag{5.15}$$

where $\beta' = 0.72167$ is a geometric factor for the cubic trap.

5.4.2 Evaluation

The purpose of developing this code was to determine whether it is possible to perform meaningful molecular dynamics simulations using the X10 FMM code developed in the previous sections. To this end we conducted initial scaling experiments for a realistic simulation of FTICR-MS over short time scales (10^2–10^5 timesteps).

The Penning trap is a 5 mm cube with a 4.7 T magnetic field and a 1 V trapping field. Initially there is a single ion cylindrical ion cloud composed of two species of similar mass/charge ratio: glutamine ($m/q = 147.07698$) and lysine ($m/q = 147.11336$). The simulation consists of timesteps of length 25 ns, using FMM evaluation of electrostatic interactions with $p = 8$.

We ran simulations with varying numbers of particles for ion clouds of amino acids of similar mass/charge ratios. Table 5.4 shows strong scaling of computation time per timestep, FMM evaluation and tree construction times for varying number of particles and maximum tree depth.

Table 5.4: Amino acids in ANU mass spectrometer: timings for one place (8 cores) to 64 places (512 cores) on *Raijin*, ($p = 8$).

	time per cycle (s)					
	$n = 10^4$, $D_{\max} = 5$			$n = 10^5$, $D_{\max} = 6$		
places	**total**	**evaluate**	**tree**	**total**	**evaluate**	**tree**
1	0.144	0.136	0.0071	2.62	2.56	0.052
2	0.0916	0.0850	0.0060	1.46	1.38	0.072
4	0.0762	0.0691	0.0067	1.08	0.990	0.088
8	0.0727	0.0629	0.0095	0.950	0.851	0.089
16	0.0671	0.0550	0.012	0.769	0.679	0.088
32	0.0726	0.0554	0.017	0.634	0.549	0.085
64	0.0758	0.0590	0.019	0.573	0.479	0.093

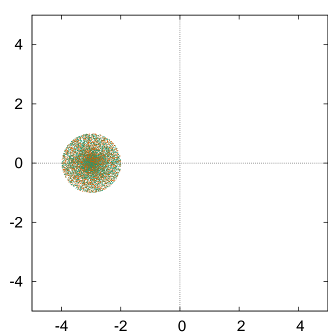
Total time reduces significantly going from 1 to 4 places for both small ($n = 10^4$) and large ($n = 10^5$) problem sizes, due to a substantial reduction in evaluation time. Tree construction time does not reduce, which contrasts with the good strong scaling of tree construction previously observed in §5.3.2.4; the major difference here is the highly non-uniform distribution of the particles, which requires a deeper tree and many more empty boxes, which in our implementation generates a greater volume of communication between places.

The ion frequencies measured in our simulation differ from the 'ideal' frequencies predicted by the modified cyclotron equation (5.13), which is to be expected as the cyclotron equation does not take ion interactions into account. Table 5.5 shows the predicted and measured frequencies from the simulation of a packet of 5000 glutamine and 5000 lysine ions.

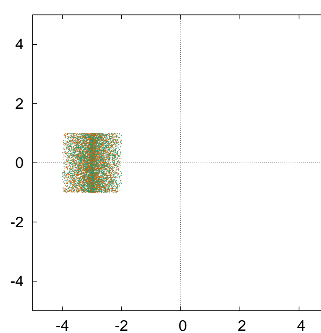
Finally, as a qualitative evaluation, we consider the high-level structure of the ion clouds after a number of revolutions. Figure 5.15 shows the ion cloud evolution in the first 2.5 ms for a simulation of 10,000 ions. After 2.5 ms, the clouds have separated and are beginning to form a comet shape. This shape has been observed previously in other simulations [Nikolaev et al., 2007] and attributed to ion–ion interactions.

Table 5.5: Amino acids in ANU mass spectrometer: predicted and measured frequencies

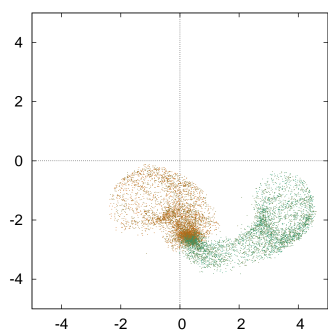
	Simulation parameters				Frequency (Hz)	
species	**charge**	**mass**	r (mm)	v_0 (m s^{-1})	**Predicted** ν_+	**Measured** ν'_+
glutamine	1	147.07698	3.0	9.25×10^3	489,779	489,540
lysine	1	147.11336	3.0	9.25×10^3	489,658	489,407



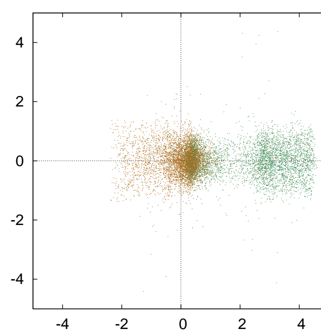
(a) time 0: X-Y projection



(b) time 0: X-Z projection



(c) time 2.5ms: X-Y projection



(d) time 2.5ms: X-Z projection

Figure 5.15: FTICR-MS ion cloud evolution in first 2.5 ms of simulation: packet of 5000 lysine and 5000 glutamine ions.

5.5 Summary

This chapter described the use of the X10 programming language to implement molecular dynamics applications. Three different algorithms were considered for the calculation of long-range electrostatic interactions, which vary in computational complexity and locality of interaction. Of the algorithms considered, the fast multipole method exhibited superior scalability due to the highly localized nature of interactions in this method. The X10 programming model supported the expression of the distributed tree structure for the FMM and the creation and synchronization of distributed parallel activities operating on nodes within the tree. Work stealing was effective in balancing the load between cores within a place, however load balancing between places was controlled by the programmer through domain decomposition of distributed data structures. The FMM was applied to the simulation of interactions between charged particles in a mass spectrometer. Load imbalance between places limited the scalability of this simulation.

Chapter 6

Conclusion

This work reported in this thesis was part of an ongoing co-design effort involving researchers at IBM and ANU, aimed at improving the expressiveness and performance of the X10 language through its use to develop two significant computational chemistry application codes. These were the first significant X10 applications created outside of the X10 development team, and are representative of a broad range of scientific and engineering applications, and have been complemented by other recent application development elsewhere. We proposed and implemented improvements to the X10 language and runtime libraries for managing and visualizing the data locality of parallel tasks, communication using active messages, and efficient implementation of distributed arrays. These improvements were demonstrated in the context of the application examples.

The performance results presented in this thesis show that X10 programs can achieve performance comparable to established programming languages when run on a single core. More importantly, X10 programs can achieve high parallel efficiency on a multithreaded architecture for parallel tasks that are created in a divide-and-conquer style and make appropriate use of worker-local data. For distributed memory architectures, X10 supports the use of active messages to construct local, asynchronous communication patterns which exhibit superior scaling compared to global, synchronous patterns.

The scalability of the application codes presented in this thesis was limited by the lack of efficient collective communication mechanisms. The benefit of such mechanisms has long been established in MPI, and their incorporation into the APGAS model will be necessary to achieve high scalability for many scientific codes. While this thesis did not provide a comprehensive solution to this problem, the concept of *collective active messages* presented in chapter 3 may provide a framework for such a solution.

Increasing programmer productivity was a primary design goal of X10. The application codes presented in this thesis are generally shorter and, I believe, easier to understand than equivalent codes using the fragmented model currently dominant

in HPC[1]. While simplicity and clarity of code are important factors in productivity, equally important are the tools available to developers. Good compilers, debuggers and profilers can contribute as much to developer productivity as the language itself. For X10 to achieve its productivity goal will require significant further investment in tooling.

X10 was originally conceived in 2003, in the early days of multicore computing. In terms of high-performance computing, systems with multiple cores per socket only entered the TOP500 list for the first time in June 2002 and single-core systems made up the majority of the list until June 2007 [TOP500]. The decision to exploit multicore parallelism through a work-stealing runtime has been proven to be effective, as demonstrated by the performance results on 8- and 16-core systems presented in this thesis. Whereas a decade ago the landscape of HPC was dominated by a single basic model – the cluster of commodity processors – the landscape today is highly varied. The fastest computers now manifest architectural heterogeneity through the use of accelerators such as GPUs and Intel's Xeon Phi; architectures featuring heterogeneous cores within a single chip will also play an important role in the future. Applying X10's APGAS model to such varied systems is a challenging and important area for future research.

Data movement is a critical factor in both performance and energy use in modern computing, which means that data locality must be a primary concern for programmers. While the APGAS model explicitly represents data locality, X10's *places* are too simple to adequately represent the deep memory hierarchies which are a feature of many HPC architectures. The task visualization approach presented in chapter 3 of this thesis could be viewed as a method to discover locality information that is hidden from the programmer in the simple model of places. A more nuanced representation of locality such as *hierarchical place trees* [Yan et al., 2010] or the *hierarchical locales* introduced in version 1.8 of the Chapel language [Chamberlain et al., 2013] would better support programmers in controlling data movement.

The success or failure of a programming language is ultimately determined by its use to develop effective computing applications, which is affected by a wide range of issues. The qualities of expressiveness and performance considered in this thesis are only two such issues; others include vendor strategies, user communities, the availability of high-quality code libraries, programming and analysis tools and the existence of highly-visible flagship applications. The contributions in this thesis have been presented in the context of a single programming language, X10, and risk being lost if that particular language fails to have widespread impact. To mitigate against this risk, it will be necessary to show how these contributions could be applied to other languages.

An area of research not touched on in this thesis is the use of domain-specific languages, which allow the programmer to express the problem directly in terms of concepts from the scientific domain in question. Automated methods are then used to generate efficient parallel code from this high-level description. For exam-

[1] sequential language + OpenMP + MPI

ple, in quantum chemistry, the Tensor Contraction Engine [Baumgartner et al., 2005] supports the generation of efficient array computations from high-level tensor expressions (which are generalized matrix multiplications). Domain-specific languages may be preferable to general-purpose languages in both expressiveness and performance, for particular applications. In this respect, X10 could be used as a framework for the implementation of domain-specific languages, rather than through direct use by application programmers.

6.1 Future Work

The implementation of the X10RT communication library on top of MPI ensures that X10 programs are portable to a wide range of distributed computer architectures. X10 has a very general model of active messages which does not map well to one-sided communications; therefore X10 messages have been implemented using two-sided MPI communications. Certain classes of active messages, for example, remote memory accesses and the collective active messages **`finish`** / **`ateach`**, are suitable for implementation on top of the new MPI-3 standard for one-sided communications, which would provide better application performance.

Local-synchronization algorithms, such as the ghost region update algorithm presented in chapter 3 and the tree traversal used in the FMM in chapter 5, would benefit from efficient implementation of atomic blocks. In general, atomic blocks are a natural fit to transactional memory; some experimental work has already been done towards an implementation using software transactional memory. Hardware support for transactional memory in Blue Gene/Q [Haring et al., 2012] and Intel's Haswell architecture [Intel, 2013a] could also be used to improve the efficiency of atomic blocks.

The exploitation of locality is the key insight in both linear-scaling methods and the APGAS programming model; their combination represents an attractive opportunity for future co-design efforts.

Appendices

Appendix A

Evaluation Platforms

Evaluation was performed on five different parallel machines:

- a typical desktop machine, containing a quad-core 3.4 GHz Sandy Bridge Core i7-2600 with 8 GB DDR3-1333 memory;
- a loosely coupled cluster machine: the *Vayu* Oracle/Sun Constellation cluster installed at the NCI National Facility at the Australian National University. Each node of *Vayu* is a Sun X6275 blade, containing two quad-core 2.93 GHz Intel Nehalem CPUs, 24 GB DDR3-1333 memory and on-board QDR InfiniBand;
- a highly-multithreaded cluster machine: the *Raijin* Fujitsu Primergy cluster installed at the NCI National Facility at the Australian National University. Each node of *Raijin* contains two eight-core 2.6 GHz Intel Xeon Sandy Bridge CPUs, 32 GB DDR3-1600 memory and on-board FDR InfiniBand;
- a tightly integrated system with a custom interconnect: the *Watson 4P* Blue Gene/P system at IBM Watson Research Center. Each Blue Gene/P compute node contains 4 PowerPC 450 compute cores running at 850 MHz with 8 MB of shared L3 cache and 2 GB of DDR-2 memory.
- a highly-multithreaded, tightly integrated system with a custom interconnect: the *Watson 2Q* Blue Gene/Q system at IBM Watson Research Center. Each Blue Gene/Q compute node contains 16 PowerPC A2 1.6 GHz compute cores (with an additional core for operating system services) with 32 MB of shared L2 cache and 16 GB DDR-3 memory [Haring et al., 2012].

For all reported results, the Native (C++ backend) version of X10 2.4 was used. One multithreaded X10 place was created per socket, with `X10_NTHREADS` (the number of worker threads) equal to the number of cores. Thus `X10_NTHREADS`=4 was used for the quad-core CPUs of *Vayu*, `X10_NTHREADS`=8 was used for the eight-core CPUs of *Raijin*, and `X10_NTHREADS`=16 was used for the 16-core CPUs of *Watson 2Q*. On Blue Gene/P however it was not possible to run using multiple threads due to thread creation restrictions on that platform. Therefore four single-threaded places were created for each quad-core CPU of *Watson 4P*.

On *Vayu* and *Raijin*, Intel MPI verson 4.1.1.036 was used with the options `-binding domain=socket -perhost 2` to run one process per quad-core or eight-core socket.

List of Abbreviations

ARMCI the Aggregate Remote Memory Copy Interface

APGAS asynchronous partitioned global address space

API application programming interface

AVX Intel Advanced Vector Extensions x86 instruction set

BLAS Basic Linear Algebra Subroutines

CAF Coarray Fortran

CUDA Compute Unified Device Architecture - a programming model for heterogeneous devices from NVIDIA Corporation

DARPA the Defence Advanced Research Projects Agency - the United States agency responsible for funding research into defence technologies

DGEMM double-precision general matrix multiply - a BLAS routine to multiply dense matrices

DSYRK double-precision symmetric rank-K update - a BLAS routine to multiply a symmetric matrix by its own transpose

FFT fast Fourier transform

FMM fast multipole method

FLOP floating point operation

FTICR-MS Fourier transform ion cyclotron resonance mass spectrometry

GML X10 Global Matrix Library - a distributed linear algebra library written in X10

GPU graphics processing unit

HPC high performance computing

HPCS High Productivity Computing Systems - a DARPA funded research program running from 2002 to 2012

JVM Java virtual machine

LAPACK Linear Algebra PACKage

MD molecular dynamics

MPI Message Passing Interface - a standard application programming interface for message passing

OpenCL Open Computing Language - an open-standard programming model for heterogeneous devices supported by multiple vendors

OpenMP Open Multi-Processing - an open-standard shared memory programming model

PAMI the Parallel Active Message Interface - a proprietary communications API developed by IBM for use on high-performance interconnects

PGAS partitioned global address space

PME particle mesh Ewald method

Pthreads the Posix Threads standard

RO resolution of the Coulomb operator

RDMA remote direct memory access

SCF self-consistent field method

SIMD single instruction, multiple data

SPMD single program, multiple data

SSE Streaming SIMD Extensions x86 instruction set

TBB Intel Threading Building Blocks

UPC Unified Parallel C

Bibliography

Acar, U. A.; Blelloch, G. E.; and Blumofe, R. D., 2000. The data locality of work stealing. In *Proceedings of the 12th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '00)*, SPAA '00, 1–12. ACM, New York, NY, USA. doi:10.1145/341800.341801. (cited on page 44)

Allen, E.; Chase, D.; Hallett, J.; Luchangco, V.; Maessen, J.-W.; Ryu, S.; Guy, L. S. J.; and Tobin-Hochstadt, S., 2008. The Fortress language specification. Technical report, Sun Microsystems. http://research.sun.com/projects/plrg/fortress.pdf. (cited on pages 22 and 23)

Anderson, E.; Bai, Z.; Bischof, C.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; and Ostrouchov, S., 1995. LAPACK users' guide, release 2.0. Technical report. (cited on pages 20 and 37)

ANUChem. The ANUChem collection of computational chemistry codes. http://cs.anu.edu.au/~Josh.Milthorpe/anuchem.html. (cited on pages 62 and 83)

Asadchev, A. and Gordon, M. S., 2012. New multithreaded hybrid CPU/GPU approach to Hartree–Fock. *Journal of Chemical Theory and Computation*, 8, 11 (2012), 4166–4176. doi:10.1021/ct300526w. (cited on page 64)

Asanovic, K.; Bodik, R.; Catanzaro, B.; Gebis, J.; Husbands, P.; Keutzer, K.; Patterson, D.; Plishker, W.; Shalf, J.; Williams, S.; and Yelick, A., 2006. The landscape of parallel computing research: A view from Berkeley. Technical report, University of Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. (cited on pages 35 and 36)

Ayguade, E.; Copty, N.; Duran, A.; Hoeflinger, J.; Lin, Y.; Massaioli, F.; Teruel, X.; Unnikrishnan, P.; and Zhang, G., 2009. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20, 3 (2009), 404–418. doi:10.1109/TPDS.2008.105. (cited on page 11)

Balay, S.; Brown, J.; Buschelman, K.; Eijkhout, V.; Gropp, W. D.; Kaushik, D.; Knepley, M. G.; McInnes, L. C.; Smith, B. F.; and Zhang, H., 2011. PETSc users manual. Technical Report ANL-95/11 - Revision 3.2, Argonne National Laboratory. (cited on page 57)

Barriuso, R. and Knies, A., 1994. SHMEM user's guide for C. Technical report, Cray Research Inc. (cited on page 9)

Baumgartner, G.; Auer, A.; Bernholdt, D. E.; Bibireata, A.; Choppella, V.; Cociorva, D.; Gao, X.; Harrison, R. J.; Hirata, S.; Krishnamoorthy, S.; Krishnan, S.; Lam, C.-C.; Lu, Q.; Nooijen, M.; Pitzer, R. M.; Ramanujma, P. S.; and Sibiryakov, A., 2005. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93, 2 (February 2005). (cited on page 115)

Berendsen, H., 2007. *Simulating the physical world*. Cambridge University Press. ISBN 978-0-521-83527-5. (cited on pages 7 and 37)

Birdsall, C. and Langdon, A., 1985. *Plasma Physics via Computer Simulation*. McGraw-Hill, New York. (cited on pages 106, 107, and 108)

Blackford, L.; Demmel, J.; Dongarra, J.; Duff, I.; Hammarling, S.; Henry, G.; Heroux, M.; Kaufman, L.; Lumsdaine, A.; Petitet, A.; Pozo, R.; and Remington, 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28, 2 (2002), 135–151. doi:10.1145/567806.567807. (cited on pages 20 and 37)

Blackford, L. S.; Choi, J.; Cleary, A.; Demmel, J.; Dhillon, I.; Dongarra, J.; Hammarling, S.; Henry, G.; Petitet, A.; Stanley, K.; Walker, D.; and Whaley, R. C., 1996. ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 5–5. doi:10.1109/SUPERC.1996.183513. (cited on pages 36 and 37)

Blumofe, R. and Leiserson, C., 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46 (Sep 1999). doi:10.1145/324133.324234. (cited on page 12)

Bonachea, D., 2002. GASNet specification, v1. Technical Report UCB/CSD-02-1207, University of California, Berkeley. (cited on page 9)

Bonachea, D. and Duell, J., 2004. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1, 1-3 (Aug. 2004), 91–99. doi:10.1504/IJHPCN.2004.007569. (cited on page 8)

Boris, J., 1970. Relativistic plasma simulation-optimization of a hybrid code. In *Proceedings of the Fourth Conference on Numerical Simulation of Plasmas*, 3–67. (cited on page 107)

Boyd, J. P., 2001. *Chebyshev and Fourier spectral methods*. Dover Publications. ISBN 0-486-41183-4. (cited on page 38)

Browne, S.; Dongarra, J.; Garner, N.; Ho, G.; and Mucci, P., 2000. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14, 3 (2000), 189–204. doi:10.1177/109434200001400303. (cited on page 85)

Cavé, V.; Zhao, J.; Shirako, J.; and Sarkar, V., 2011. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, 51–61. ACM, New York, NY, USA. doi:10.1145/2093157.2093165. (cited on page 21)

Chamberlain, B., 2001. *The Design and Implementation of a Region-Based Parallel Language*. Ph.D. thesis, University of Washington. http://www.cs.washington.edu/research/zpl/papers/data/Chamberlain01Design.pdf. (cited on page 18)

Chamberlain, B.; Callahan, D.; and Zima, H., 2007. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21 (2007), 291–312. doi:10.1177/1094342007078442. (cited on pages 8 and 21)

Chamberlain, B.; Choi, S.-E.; Deitz, S.; Iten, D.; and Litvinov, V., 2011. Authoring user-defined domain maps in Chapel. In *Proceedings of the Cray User Group*. http://chapel.cray.com/publications/cug11-final.pdf. (cited on page 22)

Chamberlain, B. L.; Choi, S.-E.; Dumler, M.; Hildebrandt, T.; Iten, D.; Litvinov, V.; and Titus, G., 2013. The State of the Chapel Union. In *Proceedings of the Cray User Group 2013*. (cited on page 114)

Chandramowlishwaran, A.; Choi, J.; Madduri, K.; and Vuduc, R., 2012. Towards a communication optimal fast multipole method and its implications at exascale. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*, 182–184. doi:10.1145/2312005.2312039. (cited on page 95)

Chandramowlishwaran, A.; Madduri, K.; and Vuduc, R., 2010. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*. doi:10.1109/SC.2010.19. (cited on page 85)

Chapel, 2014. Chapel language specification version 0.96. Technical report, Cray Inc. (cited on page 21)

Charles, P.; Grothoff, C.; Saraswat, V.; Donawa, C.; Kielstra, A.; Ebcioğlu, K.; von Praun, C.; and Sarkar, V., 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, 519–538. doi:10.1145/1094811.1094852. (cited on pages 17 and 54)

Chavarría-Miranda, D.; Krishnamoorthy, S.; and Vishnu, A., 2012. Global futures: A multithreaded execution model for global arrays-based applications. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, 393 –401. doi:10.1109/CCGrid.2012.105. (cited on page 20)

Chen, D.; Eisley, N.; Heidelberger, P.; Senger, R.; Sugawara, Y.; Kumar, S.; Salapura, V.; Satterfield, D.; Steinmacher-Burow, B.; and Parker, J., 2011. The IBM Blue Gene/Q interconnection network and message unit. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC),*, 1–10. doi:10.1145/2063384.2063419. (cited on page 49)

Colella, P., 2004. Defining software requirements for scientific computing. presentation. (cited on page 35)

Colella, P.; Graves, D. T.; Ligocki, T. J.; Martin, D. F.; Modiano, D.; Serafini, D. B.; and Van Straalen, B., 2012. *Chombo Software Package for AMR Applications-Design Document*. 3.1. Applied Numerical Algorithms Group, Lawrence Berkeley National Laboratory, Berkeley, CA. (cited on page 36)

Dachsel, H., 2006. Fast and accurate determination of the Wigner rotation matrices in the fast multipole method. *Journal of Chemical Physics*, 124, 14 (Apr 2006), 144115. doi:10.1063/1.2194548. (cited on page 92)

Dagum, L. and Menon, R., 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5 (1998), 46. doi:10.1109/99.660313. (cited on page 11)

Darden, T.; York, D.; and Pedersen, L., 1993. Particle mesh Ewald: An N log(N) method for ewald sums in large systems. *Journal of Chemical Physics*, 98, 12 (Jun. 1993), 10089–10092. doi:10.1063/1.464397. (cited on page 29)

Dayarathna, M.; Houngkaew, C.; Ogata, H.; and Suzumura, T., 2012a. Scalable performance of ScaleGraph for large scale graph analysis. In *Proceedings of the 19th International Conference on High Performance Computing (HiPC 2012)*, 1–9. doi:10.1109/HiPC.2012.6507498. (cited on page 37)

Dayarathna, M.; Houngkaew, C.; and Suzumura, T., 2012b. Introducing Scale-Graph: an X10 library for billion scale graph analytics. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, 6:1–6:9. ACM, New York, NY, USA. doi:10.1145/2246056.2246062. (cited on page 37)

Deserno, M. and Holm, C., 1998. How to mesh up Ewald sums. II. an accurate error estimate for the particle–particle–particle–mesh algorithm. *Journal of Chemical Physics*, 109, 18 (Nov. 1998), 7694–7701. doi:10.1063/1.477415. (cited on page 31)

DiStasio, R. A.; Jung, Y.; and Head-Gordon, M., 2005. A resolution-of-the-identity implementation of the local triatomics-in-molecules model for second-order Møller–Plesset perturbation theory with application to alanine tetrapeptide conformational

energies. *Journal of Chemical Theory and Computation*, 1, 5 (Sep. 2005), 862–876. doi:10.1021/ct050126s. (cited on page 73)

Dombroski, J. P.; Taylor, S. W.; and Gill, P. M. W., 1996. KWIK: Coulomb energies in $\mathcal{O}(N)$ work. *Journal of Physical Chemistry*, 100, 15 (Jan. 1996), 6272–6276. doi:10.1021/jp952841b. (cited on page 27)

Dongarra, J.; Beckman, P.; Moore, T.; Aerts, P.; Aloisio, G.; Andre, J.-C.; Barkai, D.; Berthou, J.-Y.; Boku, T.; Braunschweig, B.; Cappello, F.; Chapman, B.; Chi, X.; Choudhary, A.; Dosanjh, S.; Dunning, T.; Fiore, S.; Geist, A.; Gropp, B.; Harrison, R.; Hereld, M.; Heroux, M.; Hoisie, A.; Hotta, K.; Jin, Z.; Ishikawa, Y.; Johnson, F.; Kale, S.; Kenway, R.; Keyes, D.; Kramer, B.; Labarta, J.; Lichnewsky, A.; Lippert, T.; Lucas, B.; MacCabe, B.; Matsuoka, S.; Messina, P.; Michielse, P.; Mohr, B.; Mueller, M.; Nagel, W.; Nakashima, H.; Papka, M.; Reed, D.; Sato, M.; Seidel, E.; Shalf, J.; Skinner, D.; Snir, M.; Sterling, T.; Stevens, R.; Streitz, F.; Sugar, B.; Sumimoto, S.; Tang, W.; Taylor, J.; Thakur, R.; Trefethen, A.; Valero, M.; van der Steen, A.; Vetter, J.; Williams, P.; Wisniewski, R.; and Yelick, K., 2011. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25, 1 (2011). doi:10.1177/1094342010391989. (cited on page 17)

Dongarra, J.; Graybill, R.; Harrod, W.; Lucas, R.; Lusk, E.; Luszczek, P.; McMahon, J.; Snavely, A.; Vetter, J.; Yelick, K.; Alam, S.; Campbell, R.; Carrington, L.; Chen, T.-Y.; Khalili, O.; Meredith, J.; and Tikir, M., 2008. DARPA's HPCS program: History, models, tools, languages. In *Advances in Computers* (Ed. Marvin V. Zelkowitz), vol. Volume 72, 1–100. Elsevier. ISBN 0065-2458. doi:10.1016/S0065-2458(08)00001-6. (cited on pages 3 and 17)

Dyczmons, V., 1973. No N^4-dependence in the calculation of large molecules. *Theoretica chimica acta*, 28, 3 (Sep. 1973), 307–310. doi:10.1007/BF00533492. (cited on page 26)

Eleftheriou, M.; Moreira, J.; Fitch, B.; and Germain, R., 2003. A volumetric FFT for BlueGene/L. In *Proceedings of the 10th International Conference on High Performance Computing (HiPC 2003)*, 194–203. (cited on page 89)

Epperly, T.; Kumfert, G.; Dahlgren, T.; Ebner, D.; Leek, J.; Prantl, A.; and Kohn, S., 2011. High-performance language interoperability for scientific computing through Babel. *International Journal of High Performance Computing Applications*, (2011). doi:10.1177/1094342011414036. (cited on page 22)

Essmann, U.; Perera, L.; Berkowitz, M.; Darden, T.; Lee, H.; and Pedersen, L., 1995. A smooth particle mesh Ewald method. *Journal of Chemical Physics*, 103 (1995), 8577–8593. doi:10.1063/1.470117. (cited on pages 29 and 87)

exaFMM. https://bitbucket.org/rioyokota/exafmm-dev. Accessed: Dec 12, 2012. (cited on page 98)

Faraj, A.; Kumar, S.; Smith, B.; Mamidala, A.; and Gunnels, J., 2009. MPI collective communications on the Blue Gene/P supercomputer: Algorithms and optimizations. In *Proceedings of the 17th IEEE symposium on High Performance Interconnects (HOTI 2009)*, 63–72. doi:10.1109/HOTI.2009.12. (cited on page 49)

Filippone, S. and Colajanni, M., 2000. PSBLAS: a library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26, 4 (Dec. 2000), 527–550. doi:10.1145/365723.365732. (cited on page 36)

Fink, S.; Knobe, K.; and Sarkar, V., 2000. Unified analysis of array and object references in strongly typed languages. In *Proceedings of the Seventh International Static Analysis Symposium (SAS 2000)*. doi:10.1007/978-3-540-45099-3_9. (cited on page 55)

Fock, V., 1930. Näherungsmethode zur Lösung des quantenmechanischen Mehrkörperproblems. *Physik*, 61 (1930), 126–148. (cited on page 24)

Frigo, M. and Johnson, S., 2005. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93, 2 (Feb 2005), 216–231. doi:10.1109/JPROC.2004.840301. (cited on pages 36 and 89)

Frisch, M. J.; Trucks, G. W.; Schlegel, H. B.; Scuseria, G. E.; Robb, M. A.; Cheeseman, J. R.; Scalmani, G.; Barone, V.; Mennucci, B.; Petersson, G. A.; Nakatsuji, H.; Caricato, M.; Li, X.; Hratchian, H. P.; Izmaylov, A. F.; Bloino, J.; Zheng, G.; Sonnenberg, J. L.; Hada, M.; Ehara, M.; Toyota, K.; Fukuda, R.; Hasegawa, J.; Ishida, M.; Nakajima, T.; Honda, Y.; Kitao, O.; Nakai, H.; Vreven, T.; Montgomery, J. A., Jr.; Peralta, J. E.; Ogliaro, F.; Bearpark, M.; Heyd, J. J.; Brothers, E.; Kudin, K. N.; Staroverov, V. N.; Kobayashi, R.; Normand, J.; Raghavachari, K.; Rendell, A.; Burant, J. C.; Iyengar, S. S.; Tomasi, J.; Cossi, M.; Rega, N.; Millam, J. M.; Klene, M.; Knox, J. E.; Cross, J. B.; Bakken, V.; Adamo, C.; Jaramillo, J.; Gomperts, R.; Stratmann, R. E.; Yazyev, O.; Austin, A. J.; Cammi, R.; Pomelli, C.; Ochterski, J. W.; Martin, R. L.; Morokuma, K.; Zakrzewski, V. G.; Voth, G. A.; Salvador, P.; Dannenberg, J. J.; Dapprich, S.; Daniels, A. D.; Farkas, O.; Foresman, J. B.; Ortiz, J. V.; Cioslowski, J.; and Fox, D. J., 2009. Gaussian 09 Revision A.1. Gaussian Inc. Wallingford CT. (cited on page 24)

Fujiwara, M.; Happo, N.; and Tanaka, K., 2010. Influence of ion-ion Coulomb interactions on FT-ICR mass spectra at a high magnetic field: A many-particle simulation using a special-purpose computer. *Journal of the Mass Spectrometry Society of Japan*, 58 (2010), 169–173. doi:10.5702/massspec.58.169. (cited on page 35)

Ganesan, N.; Bauer, B.; Patel, S.; and Taufer, M., 2011. FENZI: GPU-enabled molecular dynamics simulations of large membrane regions based on the CHARMM force field and PME. In *Proceedings of Tenth IEEE International Workshop on High Performance Computational Biology (HiCOMB 2011)*. doi:10.1109/IPDPS.2011.187. (cited on page 87)

Graph500, 2010. The Graph500 list. http://www.graph500.org/. (cited on page 37)

Greengard, L. and Rokhlin, V., 1987. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73 (1987), 325. doi:10.1016/0021-9991(87)90140-9. (cited on page 31)

Greengard, L. and Rokhlin, V., 1997. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numerica*, 6 (1997), 229. doi:10.1017/s0962492900002725. (cited on page 31)

Grove, D.; Milthorpe, J.; and Tardieu, O., 2014. Supporting array programming in X10. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14 (Edinburgh, United Kingdom, 2014), 38–43. ACM, New York, NY, USA. doi:10.1145/2627373.2627380. (cited on page 54)

Grove, D.; Tardieu, O.; Cunningham, D.; Herta, B.; Peshansky, I.; and Saraswat, V., 2011. A performance model for X10 applications: what's going on under the hood? In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, 1:1–1:8. ACM, New York, NY, USA. doi:10.1145/2212736.2212737. (cited on pages 40 and 97)

Guan, S. and Marshall, A. G., 1995. Ion traps for Fourier transform ion cyclotron resonance mass spectrometry: principles and design of geometric and electric configurations. *International Journal of Mass Spectrometry and Ion Processes*, 146/147 (1995), 261–296. (cited on pages 107 and 108)

Hadoop. http://hadoop.apache.org. Apache Software Foundation. (cited on page 36)

Hall, G. G., 1951. The molecular orbital theory of chemical valency. VIII. a method of calculating ionization potentials. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 205, 1083 (Mar. 1951), 541–552. doi:10.1098/rspa.1951.0048. (cited on page 25)

Hamada, T.; Narumi, T.; Yokota, R.; Yasuoka, K.; Nitadori, K.; and Taiji, M., 2009. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 62:1–62:12. ACM, New York, NY, USA. doi:10.1145/1654059.1654123. (cited on page 38)

Hamouda, S. S.; Milthorpe, J.; Strazdins, P. E.; and Saraswat, V., 2015. A resilient framework for iterative linear algebra applications in x10. In *Proceedings of the 16th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, PDSEC 2015. (cited on page 21)

Hansson, T.; Oostenbrink, C.; and van Gunsteren, W. F., 2002. Molecular dynamics simulations. *Current Opinion in Structural Biology*, 12, 2 (2002), 190–196. doi:10.1016/S0959-440X(02)00308-1. (cited on page 29)

Haring, R.; Ohmacht, M.; Fox, T.; Gschwind, M.; Satterfield, D.; Sugavanam, K.; Coteus, P.; Heidelberger, P.; Blumrich, M.; Wisniewski, R.; Gara, A.; Chiu, G.-T.; Boyle, P.; Chist, N.; and Kim, C., 2012. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32, 2 (2012), 48–60. doi:10.1109/MM.2011.108. (cited on pages 103, 115, and 119)

Harrison, R.; Guest, M.; Kendall, R.; Bernholdt, D.; Wong, A.; Stave, M.; Anchell, J.; Hess, A.; Littlefield, R.; Fann, G.; Nieplocha, J.; Thomas, G.; Elwood, D.; Tilson, J.; Shepard, R.; Wagner, A.; Foster, I.; Lusk, E.; and Stevens, R., 1996. Toward high-performance computational chemistry: II. A scalable self-consistent field program. *Journal of Computational Chemistry*, 17 (Jan 1996), 124–132. doi:10.1002/(SICI)1096-987X(19960115)17:1<124::AID-JCC10>3.0.CO;2-N. (cited on page 26)

Häser, M. and Ahlrichs, R., 1989. Improvements on the direct SCF method. *Journal of Computational Chemistry*, 10, 1 (1989), 104–111. doi:10.1002/jcc.540100111. (cited on page 63)

Heffelfinger, G. S., 2000. Parallel atomistic simulations. *Computer Physics Communications*, 128, 1–2 (Jun. 2000), 219–237. doi:10.1016/S0010-4655(00)00050-3. (cited on page 85)

Heroux, M. A. and Dongarra, J., 2013. Toward a new metric for ranking high performance computing systems. Technical Report SAND2013-4744, Sandia National Laboratories. (cited on page 37)

Hess, B.; Kutzner, C.; van der Spoel, D.; and Lindahl, E., 2008. GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, 4, 3 (Mar 2008), 435–447. doi:10.1021/ct700301q. (cited on pages 36, 83, and 84)

Hess, B.; van der Spoel, D.; and Lindahl, E., 2013. GROMACS user manual. Technical Report 4.6.1, University of Groningen. (cited on page 31)

Hochstein, L.; Carver, J.; Shull, F.; Asgari, S.; Basili, V.; Hollingsworth, J. K.; and Zelkowitz, M. V., 2005. Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, 35. IEEE Computer Society, Washington, DC, USA. doi:10.1109/SC.2005.53. (cited on page 17)

Hockney, R. and Eastwood, J., 1988. *Computer simulation using particles*. Institute of Physics Publishing. ISBN 0-85274-392-0. (cited on page 37)

Hoefler, T.; Siebert, C.; and Rehm, W., 2007. A practically constant-time MPI broadcast algorithm for large-scale InfiniBand clusters with multicast. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, 1–8. doi:10.1109/IPDPS.2007.370475. (cited on page 49)

Huang, C.; Lawlor, O.; and Kalé, L. V., 2003. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, 306–322. (cited on page 8)

IBM, 2012. A2 processor user's manual for Blue Gene/Q. Technical report, IBM. https://wiki.alcf.anl.gov/parts/images/c/cf/A2.pdf. (cited on page 5)

IEEE, 2008. IEEE standard for information technology- portable operating system interface (POSIX) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, (2008). doi:10.1109/IEEESTD.2008.4694976. (cited on page 10)

Intel, 2011. Intel 64 and IA-32 architectures optimization reference manual. Technical Report 248966-025, Intel Corporation. (cited on page 84)

Intel, 2013a. Intel 64 and IA-32 architectures software developer manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. (cited on pages 5 and 115)

Intel, 2013b. Intel MPI benchmarks user guide and methodology description version 3.2.3. Technical Report 320714-007EN, Intel Corporation. http://software.intel.com/sites/products/documentation/hpc/ics/imb/32/IMB_Users_Guide/IMB_Users_Guide.pdf. (cited on page 52)

Ishiyama, T.; Nitadori, K.; and Makino, J., 2012. 4.45 pflops astrophysical N-body simulation on K computer: the gravitational trillion-body problem. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 5:1–5:10. IEEE Computer Society Press, Los Alamitos, CA, USA. doi:10.1109/SC.2012.3. (cited on page 38)

Izmaylov, A. F.; Scuseria, G. E.; and Frisch, M. J., 2006. Efficient evaluation of short-range Hartree–Fock exchange in large molecules and periodic systems. *Journal of Chemical Physics*, 125, 10 (Sep. 2006), 104103. doi:doi:10.1063/1.2347713. (cited on page 28)

Ji, Y.; Liu, L.; and Yang, G., 2012. Characterization of Smith-Waterman sequence database search in X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, 2:1–2:7. ACM, New York, NY, USA. doi:10.1145/2246056.2246058. (cited on page 37)

Joyner, M.; Budimlic, Z.; Sarkar, V.; and Zhang, R., 2008. Array optimizations for parallel implementations of high productivity languages. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, 1–8. doi:10.1109/IPDPS.2008.4536185. (cited on page 54)

Kalé, L. V. and Krishnan, S., 1993. CHARM++: a portable concurrent object oriented system based on C++. *SIGPLAN Notices*, 28, 10 (Oct. 1993), 91–108. doi:10.1145/167962.165874. (cited on page 8)

Kawachiya, K.; Takeuchi, M.; Zakirov, S.; and Onodera, T., 2012. Distributed garbage collection for managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, 5:1–5:11. ACM, New York, NY, USA. doi:10.1145/2246056.2246061. (cited on page 19)

Khronos, 2012. The OpenCL specification. Technical Report 1.2, Khronos OpenCL Working Group. http://www.khronos.org/registry. (cited on page 13)

Kjolstad, F. and Snir, M., 2010. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP '10)*. doi:10.1145/1953611.1953615. (cited on page 57)

Kuck, D. J., 2004. Productivity in HPC. *International Journal of High Performance Computing Applications*, 18 (Nov 2004), 489–504. doi:10.1177/1094342004048541. (cited on page 2)

Kudin, K. and Scuseria, G., 1998. A fast multipole method for periodic systems with arbitrary unit cell geometries. *Chemical Physics Letters*, 283 (Jan 1998), 61. doi:10.1016/s0009-2614(97)01329-8. (cited on page 31)

Kumar, S.; Mamidala, A.; Faraj, D.; Smith, B.; Blocksome, M.; Cernohous, B.; Miller, D.; Parker, J.; Ratterman, J.; Heidelberger, P.; Chen, D.; and Steinmacher-Burrow, B., 2012. PAMI: a parallel active message interface for the Blue Gene/Q supercomputer. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2012)*, 763–773. doi:10.1109/IPDPS.2012.73. (cited on page 9)

Kurzak, J. and Pettitt, B. M., 2005. Massively parallel implementation of a fast multipole method for distributed memory machines. *Journal of Parallel and Distributed Computing*, 65 (2005), 870–881. doi:10.1016/j.jpdc.2005.02.001. (cited on page 95)

Lambert, C.; Darden, T.; and Board, J. J., 1996. A multipole-based algorithm for efficient calculation of forces and potentials in macroscopic periodic assemblies of particles. *Journal of Computational Physics*, 126 (Jul 1996), 274. doi:10.1006/jcph.1996.0137. (cited on page 31)

Lamport, L., 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28, 9 (1979), 690–691. doi:10.1109/TC.1979.1675439. (cited on page 15)

Lashuk, I.; Chandramowlishwaran, A.; Langston, H.; Nguyen, T.-A.; Sampath, R.; Shringarpure, A.; Vuduc, R.; Ying, L.; Zorin, D.; and Biros, G., 2009. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. doi:10.1145/1654059.1654118. (cited on pages 33 and 95)

Lea, D., 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, 36–43. ACM, New York, NY, USA. doi:10.1145/337449.337465. (cited on page 13)

Leach, F.; Kharchenko, A.; Heeren, R.; Nikolaev, E.; and Amster, I., 2009. Comparison of particle-in-cell simulations with experimentally observed frequency shifts between ions of the same mass-to-charge in Fourier transform ion cyclotron resonance mass spectrometry. *Journal of the American Society for Mass Spectrometry*, 21 (Oct 2009), 203–208. doi:10.1016/j.jasms.2009.10.001. (cited on page 35)

Leiserson, C. E., 2010. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51, 3 (Mar. 2010), 244–257. doi:10.1007/s11227-010-0405-3. (cited on pages 12 and 13)

Limpanuparb, T., 2012. *Applications of Resolutions of the Coulomb Operator in Quantum Chemistry*. Ph.D. thesis, Australian National University, Canberra, Australia. http://hdl.handle.net/1885/8879. ANU Digital Collections Repository. (cited on pages 27 and 28)

Limpanuparb, T.; Hollett, J. W.; and Gill, P. M. W., 2012. Resolutions of the Coulomb operator. VI. Computation of auxiliary integrals. *Journal of Chemical Physics*, 136, 10 (Mar. 2012), 104102. doi:doi:10.1063/1.3691829. (cited on page 62)

Limpanuparb, T.; Milthorpe, J.; and Rendell, A., 2014. Resolutions of the Coulomb operator: VIII. Parallel implementation using the modern programming language X10. *Journal of Computational Chemistry*, 35, 28 (Oct 2014), 2056–2069. doi:10.1002/jcc.23720. (cited on page 62)

Limpanuparb, T.; Milthorpe, J.; Rendell, A.; and Gill, P., 2013. Resolutions of the Coulomb operator: VII. Evaluation of long-range Coulomb and exchange matrices. *Journal of Chemical Theory and Computation*, 9, 2 (Jan 2013), 863–867. doi:10.1021/ct301110y. (cited on pages 27, 28, 62, 63, and 74)

Ltaief, H. and Yokota, R., 2012. Data-driven execution of fast multipole methods. *CoRR*, (2012). arXiv:abs/1203.0889. (cited on page 33)

Lüthi, H. P.; Mertz, J. E.; Feyereisen, M. W.; and Almlöf, J. E., 1991. A coarse-grain parallel implementation of the direct SCF method. *Journal of Computational Chemistry*, 13 (1991), 160. doi:10.1002/jcc.540130207. (cited on page 26)

Maheshwary, S.; Patel, N.; Sathyamurthy, N.; Kulkarni, A. D.; and Gadre, S. R., 2001. Structure and stability of water clusters $(H_2O)_n$, n = 8–20: An ab initio investigation. *Journal of Physical Chemistry A*, 105, 46 (Nov. 2001), 10525–10537. doi:10.1021/jp013141b. (cited on page 73)

Marshall, A.; Hendrickson, C.; and Jackson, G., 1998. Fourier transform ion cyclotron resonance mass spectrometry: A primer. *Mass Spectrometry Reviews*, 17 (Jan 1998), 1–35. doi:10.1002/(SICI)1098-2787(1998)17:1<1::AID-MAS1>3.0.CO;2-K. (cited on page 33)

Martorell, X.; Labarta, J.; Navarro, N.; and Ayguadé, E., 1996. A library implementation of the nano-threads programming model. In *Euro-Par'96 Parallel Processing*, 644–649. doi:10.1007/BFb0024761. (cited on page 10)

McCool, M.; Reinders, J.; and Robison, A., 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier. ISBN 9780123914439. (cited on page 11)

Mellor-Crummey, J.; Adhianto, L.; Scherer, W. N., III; and Jin, G., 2009. A new vision for Coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programming Models*, PGAS '09, 5:1–5:9. ACM, New York, NY, USA. doi:10.1145/1809961.1809969. (cited on page 16)

Milthorpe, J.; Ganesh, V.; Rendell, A.; and Grove, D., 2011. X10 as a parallel language for scientific computation: practice and experience. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, 1080–1088. doi:10.1109/IPDPS.2011.103. (cited on pages 39, 62, 65, 83, and 90)

Milthorpe, J. and Rendell, A., 2012. Efficient update of ghost regions using active messages. In *Proceedings of the 19th IEEE International Conference on High Performance Computing (HiPC 2012)*. doi:10.1109/HiPC.2012.6507484. (cited on pages 37, 39, 55, 83, and 97)

Milthorpe, J.; Rendell, A.; and Huber, T., 2013. PGAS-FMM: Implementing a distributed fast multipole method using the X10 programming language. *Concurrency and Computation: Practice and Experience*, (May 2013). doi:10.1002/cpe.3039. (cited on page 83)

Min, S.-J.; Iancu, C.; and Yelick, K., 2011. Hierarchical work stealing on manycore clusters. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS '11)*. Galveston Island, TX. (cited on page 16)

MPI Forum, 1994. MPI: A message-passing interface standard. Technical report, MPI Forum. http://www.mpi-forum.org/docs/mpi-10.ps. (cited on page 8)

MPI Forum, 2003. MPI-2: Extensions to the message-passing interface. Technical report, MPI Forum. http://www.mpi-forum.org/docs/mpi2-report.pdf. (cited on page 8)

MPI Forum, 2012. MPI: A message-passing interface standard version 3.0. Technical report, MPI Forum. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. (cited on page 8)

Nakashima, J. and Taura, K., 2014. MassiveThreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, 222–238. doi:10.1007/978-3-662-44471-9_10. (cited on page 10)

Nieplocha, J.; Palmer, B.; Tipparaju, V.; Krishnan, M.; Trease, H.; and Apra, E., 2006a. Advances, applications and performance of the Global Arrays shared

memory programming toolkit. *International Journal of High Performance Computing Applications*, 20 (May 2006), 203. doi:10.1177/1094342006064503. (cited on pages 17, 26, and 57)

Nieplocha, J.; Tipparaju, V.; Krishnan, M.; and Panda, D. K., 2006b. High performance remote memory access communication: the ARMCI approach. *International Journal of High Performance Computing Applications*, 20, 2 (May 2006), 233–253. doi:10.1177/1094342006064504. (cited on page 9)

Nikolaev, E.; Heeren, R.; Popov, A.; Pozdneev, A.; and Chingin, K., 2007. Realistic modeling of ion cloud motion in a Fourier transform ion cyclotron resonance cell by use of a particle-in-cell approach. *Rapid Communications in Mass Spectrometry*, 21 (2007), 3527. doi:10.1002/rcm.3234. (cited on pages 35 and 109)

Numrich, R. W. and Reid, J., 1998. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17, 2 (Aug. 1998), 1–31. doi:10.1145/289918.289920. (cited on page 16)

NVIDIA, 2013. CUDA. Technical report, NVIDIA. http://developer.nvidia.com/cuda. (cited on page 15)

Ochsenfeld, C.; Kussmann, J.; and Lambrecht, D. S., 2007. Linear-scaling methods in quantum chemistry. In *Reviews in Computational Chemistry* (Eds. K. B. Lipkowitz and T. R. Cundari), 1–82. John Wiley & Sons, Inc. ISBN 9780470116449. (cited on page 27)

OpenMP ARB, 2008. OpenMP application program interface version 3.0. Technical report, OpenMP Architecture Review Board. http://www.openmp.org/mp-documents/spec30.pdf. (cited on page 11)

OpenMP ARB, 2013. OpenMP application program interface version 4.0. Technical report, OpenMP Architecture Review Board. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. (cited on page 12)

Page, L.; Brin, S.; Motwani, R.; and Winograd, T., 1999. The PageRank citation ranking: Bringing order to the Web. Technical Report SIDL-WP-1999-0120, Stanford University. http://ilpubs.stanford.edu:8090/422/. (cited on page 20)

Palmer, B. and Nieplocha, J., 2002. Efficient algorithms for ghost cell updates on two classes of MPP architectures. In *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, 197–202. http://www.emsl.pnl.gov/docs/global/papers/ghosts.pdf. (cited on page 57)

Patacchini, L. and Hutchinson, I., 2009. Explicit time-reversible orbit integration in particle in cell codes with static homogeneous magnetic field. *Journal of Computational Physics*, 228 (2009), 2604. doi:10.1016/j.jcp.2008.12.021. (cited on pages 107 and 108)

Petitet, A.; Whaley, R. C.; Dongarra, J.; and Cleary, A., 2008. HPL - a portable implementation of the High-Performance Linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/. (cited on page 37)

Poulson, J.; Marker, B.; van de Geijn, R. A.; Hammond, J. R.; and Romero, N. A., 2013. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39, 2 (Feb. 2013), 1–24. doi:10.1145/2427023.2427030. (cited on pages 36 and 37)

Prantl, A.; Epperly, T.; Imam, S.; and Sarkar, V., 2011. Interfacing Chapel with traditional HPC programming languages. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS 2011)*. (cited on page 22)

Pronk, S.; Páll, S.; Schulz, R.; Larsson, P.; Bjelkmar, P.; Apostolov, R.; Shirts, M. R.; Smith, J. C.; Kasson, P. M.; Spoel, D. v. d.; Hess, B.; and Lindahl, E., 2013. GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, (Feb. 2013). doi:10.1093/bioinformatics/btt055. (cited on page 31)

Rahimian, A.; Lashuk, I.; Veerapaneni, S.; Chandramowlishwaran, A.; Malhotra, D.; Moon, L.; Sampath, R.; Shringarpure, A.; Vetter, J.; Vuduc, R.; Zorin, D.; and Biros, G., 2010. Petascale direct numerical simulation of blood flow on 200K cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, 1–11. IEEE Computer Society, Washington, DC, USA. doi:10.1109/SC.2010.42. (cited on page 38)

Reinders, J., 2010. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc. ISBN 9781449390860. (cited on page 13)

Ritchie, D. M. and Thompson, K., 1974. The UNIX time-sharing system. *Communications of the ACM*, 17, 7 (Jul. 1974), 365–375. doi:10.1145/361011.361061. (cited on page 7)

Robison, A.; Voss, M.; and Kukanov, A., 2008. Optimization via reflection on work stealing in TBB. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*. doi:10.1109/IPDPS.2008.4536188. (cited on page 13)

Roothaan, C. C. J., 1951. New developments in molecular orbital theory. *Reviews of Modern Physics*, 23, 2 (Apr. 1951), 69–89. doi:10.1103/RevModPhys.23.69. (cited on page 25)

Saraswat, V.; Almasi, G.; Bikshandi, G.; Cascaval, C.; Cunningham, D.; Grove, D.; Kodali, S.; Peshansky, I.; and Tardieu, O., 2010. The asynchronous partitioned global address space model. In *Proceedings of The First Workshop on Advances in Message Passing*. (cited on page 2)

SARASWAT, V.; BLOOM, B.; PESHANSKY, I.; TARDIEU, O.; AND GROVE, D., 2014. X10 language specification version 2.5. Technical report, IBM. http://x10.sourceforge.net/documentation/languagespec/x10-250.pdf. (cited on pages 17, 47, and 50)

SARASWAT, V.; KAMBADUR, P.; KODALI, S.; GROVE, D.; AND KRISHNAMOORTHY, S., 2011. Lifeline-based global load balancing. In *Proceedings of the 16th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. doi:10.1145/1941553.1941582. (cited on pages 19 and 37)

SCHERER, W. N. I.; ADHIANTO, L.; JIN, G.; MELLOR-CRUMMEY, J.; AND YANG, C., 2010. Hiding latency in Coarray Fortran 2.0. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, 14:1–14:9. ACM, New York, NY, USA. doi:10.1145/2020373.2020387. (cited on page 20)

SCHMIDT, M. W.; BALDRIDGE, K. K.; BOATZ, J. A.; ELBERT, S. T.; GORDON, M. S.; JENSEN, J. H.; KOSEKI, S.; MATSUNAGA, N.; NGUYEN, K. A.; SU, S.; WINDUS, T. L.; DUPUIS, M.; AND MONTGOMERY, J. A., 1993. General atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14 (Nov 1993), 1347–1363. doi:10.1002/jcc.540141112. (cited on page 24)

SHAH, G.; NIEPLOCHA, J.; MIRZA, J.; KIM, C.; HARRISON, R.; GOVINDARAJU, R.; GILDEA, K.; DINICOLA, P.; AND BENDER, C., 1998. Performance and experience with LAPI - a new high-performance communication library for the IBM RS/6000 SP. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998)*, 260–266. doi:10.1109/IPPS.1998.669923. (cited on page 9)

SHAO, Y.; MOLNAR, L.; JUNG, Y.; KUSSMANN, J.; OCHSENFELD, C.; BROWN, S.; GILBERT, A.; SLIPCHENKO, L.; LEVCHENKO, S.; O'NEILL, D.; DISTASIO, R.; LOCHAN, R.; WANG, T.; BERAN, G.; BESLEY, N.; HERBERT, J.; LIN, C.; VAN VOORHIS, T.; CHIEN, S.; SODT, A.; STEELE, R.; RASSOLOV, V.; MASLEN, P.; KORAMBATH, P.; ADAMSON, R.; AUSTIN, B.; BAKER, J.; BYRD, E.; DACHSEL, H.; DOERKSEN, R.; DREUW, A.; DUNIETZ, B.; DUTOI, A.; FURLANI, T.; GWALTNEY, S.; HEYDEN, A.; HIRATA, S.; HSU, C.-P.; KEDZIORA, G.; KHALLIULIN, R.; KLUNZINGER, P.; LEE, A.; LEE, M.; LIANG, W.; LOTAN, I.; NAIR, N.; PETERS, B.; PROYNOV, E.; PIENIAZEK, P.; RHEE, Y.; RITCHIE, J.; ROSTA, E.; SHERRILL, C.; SIMMONETT, A.; SUBOTNIK, J.; WOODCOCK, H.; ZHANG, W.; BELL, A.; CHAKRABORTY, A.; CHIPMAN, D.; KEIL, F.; WARSHEL, A.; HEHRE, W.; SCHAEFER, H.; KONG, J.; KRYLOV, A.; GILL, P.; AND HEAD-GORDON, M., 2013. Advances in methods and algorithms in a modern quantum chemistry program package. *Physical Chemistry Chemical Physics*, 8 (Jan 2013). doi:10.1039/B517914A. (cited on pages 24 and 73)

SHAW, D.; DROR, R.; SALMON, J.; GROSSMAN, J.; MACKENZIE, K.; BANK, J.; YOUNG, C.; DENEROFF, M.; BATSON, B.; BOWERS, K.; CHOW, E.; EASTWOOD, M.; IERARDI, D.; KLEPEIS, J.; KUSKIN, J.; LARSON, R.; LINDORFF-LARSEN, K.; MARAGAKIS, P.; MORAES, M.; PIANA, S.; SHAN, Y.; AND TOWLES, B., 2009. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance*

Computing Networking, Storage and Analysis, SC '09. doi:10.1145/1654059.1654099. (cited on page 29)

Shet, A.; Elwasif, W.; Harrison, R.; and Bernholdt, D., 2008. Programmability of the HPCS languages: A case study with a quantum chemistry kernel (extended version). Technical report, Oak Ridge National Laboratory. http://www.csm.ornl.gov/~anish12/ldoc9885_hips_tr.pdf. (cited on pages 26 and 64)

Shet, A. G.; Tipparaju, V.; and Harrison, R. J., 2009. Asynchronous programming in UPC: a case study and potential for improvement. In *1st Workshop on Asynchrony in the PGAS Programming Model (APGAS)*. (cited on page 20)

Shinnar, A.; Cunningham, D.; Saraswat, V.; and Herta, B., 2012. M3R: increased performance for in-memory Hadoop jobs. *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB 2012)*, 5, 12 (Aug. 2012), 1736–1747. doi:10.14778/2367502.2367513. (cited on page 37)

Shirako, J. and Sarkar, V., 2010. Hierarchical phasers for scalable synchronization and reductions in dynamic parallelism. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, 1–12. doi:10.1109/IPDPS.2010.5470414. (cited on page 21)

Springel, V.; Yoshida, N.; and White, S. D., 2001. GADGET: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6, 2 (Apr. 2001), 79–117. doi:10.1016/S1384-1076(01)00042-2. (cited on page 36)

Sundar, H.; Sampath, R. S.; Adavani, S. S.; Davatzikos, C.; and Biros, G., 2007. Low-constant parallel algorithms for finite element simulations using linear octrees. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 25:1–25:12. ACM, New York, NY, USA. doi:10.1145/1362622.1362656. (cited on pages 33 and 36)

Sundar, H.; Sampath, R. S.; and Biros, G., 2008. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30, 5 (Jan. 2008), 2675–2708. doi:10.1137/070681727. (cited on page 33)

Susukita, R.; Ebisuzaki, T.; Elmegreen, B. G.; Furusawa, H.; Kato, K.; Kawai, A.; Kobayashi, Y.; Koishi, T.; McNiven, G. D.; Narumi, T.; and Yasuoka, K., 2003. Hardware accelerator for molecular dynamics: MDGRAPE-2. *Computer Physics Communications*, 155, 2 (2003), 115 – 131. doi:10.1016/S0010-4655(03)00349-7. (cited on page 29)

Szabo, A. and Ostlund, N. S., 1989. *Modern Quantum Chemistry*. McGraw-Hill, New York. (cited on pages 24 and 25)

Tardieu, O.; Herta, B.; Cunningham, D.; Grove, D.; Kambadur, P.; Saraswat, V.; Shinnar, A.; Takeuchi, M.; and Vaziri, M., 2014. X10 and APGAS at petascale. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '14)*, 53–66. ACM, New York, NY, USA. doi:10.1145/2555243.2555245. (cited on page 47)

TARDIEU, O.; LIN, H. B.; AND WANG, H., 2012. A work-stealing scheduler for X10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. doi:10.1145/2370036.2145850. (cited on pages 19 and 40)

TAURA, K.; NAKASHIMA, J.; YOKOTA, R.; AND MARUYAMA, N., 2012. A task parallel implementation of fast multipole methods. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 617–625. doi:10.1109/SC.Companion.2012.86. (cited on page 32)

TERUEL, X.; KLEMM, M.; LI, K.; MARTORELL, X.; OLIVIER, S. L.; AND TERBOVEN, C., 2013. A proposal for task-generating loops in OpenMP. In *Proceedings of the 9th International Workshop on OpenMP (IWOMP 2013)*. doi:10.1007/978-3-642-40698-0_1. (cited on page 11)

TOP500. Top 500 supercomputer sites. http://www.top500.org. (cited on pages 37 and 114)

UFIMTSEV, I. S. AND MARTÍNEZ, T. J., 2008. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. *Journal of Chemical Theory and Computation*, 4, 2 (2008), 222–231. doi:10.1021/ct700268q. (cited on page 64)

UPC CONSORTIUM, 2005. UPC language specification v1.2. Technical report, Lawrence Berkeley National Laboratory. http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf. (cited on page 15)

VALIEV, M.; BYLASKA, E.; GOVIND, N.; KOWALSKI, K.; STRAATSMA, T.; VAN DAM, H.; WANG, D.; NIEPLOCHA, J.; APRA, E.; WINDUS, T.; AND DE JONG, W., 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181 (May 2010), 1477. doi:10.1016/j.cpc.2010.04.018. (cited on pages 17, 24, and 81)

VLADIMIROV, G.; HENDRICKSON, C.; BLAKNEY, G.; MARSHALL, A.; HEEREN, R.; AND NIKOLAEV, E., 2011. Fourier transform ion cyclotron resonance mass resolution and dynamic range limits calculated by computer modeling of ion cloud motion. *Journal of the American Society for Mass Spectrometry*, 23 (Dec 2011), 375. doi:10.1007/s13361-011-0268-8. (cited on page 35)

VON EICKEN, T.; CULLER, D. E.; GOLDSTEIN, S. C.; AND SCHAUSER, K. E., 1992. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual International Symposium on Computer Architecture (ISCA '92)*, 256–266. ACM, New York, NY, USA. doi:10.1145/139669.140382. (cited on pages 9 and 46)

WARREN, M. AND SALMON, J., 1992. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, 570–576. doi:10.1109/SUPERC.1992.236647. (cited on page 92)

Wheeler, K. B.; Murphy, R. C.; and Thain, D., 2008. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*. doi:10.1109/IPDPS.2008.4536359. (cited on page 10)

White, C. and Head-Gordon, M., 1994. Derivation and efficient implementation of the fast multipole method. *Journal of Chemical Physics*, 101, 8 (1994), 6593–6605. doi:10.1063/1.468354. (cited on pages 31 and 32)

White, C. and Head-Gordon, M., 1996. Rotating around the quartic angular momentum barrier in fast multipole method calculations. *Journal of Chemical Physics*, 105, 12 (Nov 1996). doi:10.1063/1.472369. (cited on page 92)

White, C. A.; Johnson, B. G.; Gill, P. M.; and Head-Gordon, M., 1996. Linear scaling density functional calculations via the continuous fast multipole method. *Chemical Physics Letters*, 253, 3–4 (May 1996), 268–278. doi:10.1016/0009-2614(96)00175-3. (cited on page 27)

Yan, Y.; Zhao, J.; Guo, Y.; and Sarkar, V., 2010. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, 172–187. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-642-13374-9_12. (cited on pages 21 and 114)

Yelick, K.; Bonachea, D.; Chen, W.-Y.; Colella, P.; Datta, K.; Duell, J.; Graham, S.; Hargrove, P.; Hilfinger, P.; Husbands, P.; Iancu, C.; Kamil, A.; Nishtala, R.; Su, J.; Welcome, M.; and Wen, T., 2007a. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel Symbolic Computation*, PASCO '07, 24–32. doi:10.1145/1278177.1278183. (cited on page 15)

Yelick, K.; Hilfinger, P.; Graham, S.; Bonachea, D.; Su, J.; Kamil, A.; Datta, K.; Colella, P.; and Wen, T., 2007b. Parallel languages and compilers: Perspective from the Titanium experience. *International Journal of High Performance Computing Applications*, 21, 3 (Aug. 2007), 266–290. doi:10.1177/1094342007078449. (cited on pages 16 and 54)

Yelick, K.; Semenzato, L.; Pike, G.; Miyamoto, C.; Liblit, B.; Krishnamurthy, A.; Hilfinger, P.; Graham, S.; Gay, D.; Colella, P.; and Aiken, A., 1998. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10, 11-13 (1998), 825–836. doi:10.1002/(SICI)1096-9128(199809/11)10:11/13<825::AID-CPE383>3.0.CO;2-H. (cited on page 16)

Ying, L.; Biros, G.; and Zorin, D., 2004. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196 (Jan 2004), 591. doi:10.1016/j.jcp.2003.11.021. (cited on page 31)

Ying, L.; Biros, G.; Zorin, D.; and Langston, H., 2003. A new parallel kernel-independent fast multipole method. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 14. doi:10.1145/1048935.1050165. (cited on page 33)

Yokota, R., 2013. An FMM based on dual tree traversal for many-core architectures. *Journal of Algorithms & Computational Technology*, 7, 3 (Sep. 2013), 301–324. doi:10.1260/1748-3018.7.3.301. (cited on pages 31, 32, 83, 85, 92, 98, and 100)