PERFORMANCE OPTIMIZATION OF SCIENTIFIC APPLICATIONS ON EMERGING ARCHITECTURES

by

Hikmet Dursun

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

May 2012

Dedication

To my family
Akif, Havva and Hilal Dursun

Acknowledgments

I am highly indebted to my advisor, Professor Aiichiro Nakano for his tremendous contributions to my doctoral research. His infinite source of motivation and creativity was the driving force of my Ph.D. research at the University of Southern California (USC). Beyond research, it was also a great pleasure to know him personally and discuss various topics from a multitude of disciplines. My personal and professional development has significantly benefited from these discussions.

I wish to express my gratitude to Professor Priya Vashishta and Professor Rajiv Kalia at USC, and Professor Fuyuki Shimojo from Kumamoto University for laying the groundwork on which this work is based. I thank Dr. Darren Kerbyson for his financial support and supervision during 2008 summer that I spent at Los Alamos National Laboratory (LANL). I thank Dr. Kevin Barker and Dr. Adolfy Hoisie from Pacific Northwest National Laboratory, and Dr. Scott Pakin from LANL for their collaboration on performance modeling and analysis topics. I thank Dr. Robert Lucas, Dr. Pedro Diniz and Dr. Jacqueline Chame from the Information Sciences Institute at USC, and Professor Mary Hall from the University of Utah for their contributions to our research on performance tuning technologies. I wish to express my gratitude to the members of

Collaboratory for Advanced Computing and Simulations at USC for providing an academically challenging environment that made this work possible. I thank the Department of Computer Science, and the Department of Physics and Astronomy at USC for giving me the chance to pursue my Ph.D. I would like to thank the Argonne National Laboratory and Lawrence Livermore National Laboratory for the use of their BlueGene/P and BlueGene/L supercomputers. I thank the United States Department of Energy and Department of Defense for funding my research.

I cannot thank Dr. Özgür Kalenci and Dr. Mert Gürbüzbalaban enough, whose enduring friendship and highly inspirational discussions helped me shape my academic career so far. I owe an equally large debt of gratitude to Mr. Mustafa Öncel and Mr. Şerafettin Yılmaz, who greatly stimulated my intellectual curiosity to pursue this doctoral journey. Last but not the least, I thank my family and friends for their love, encouragement and patience during my studies. None of my achievements would have been possible without their support.

Table of Contents

List of Tables

List of Figures

Abstract

The shift to many-core architecture design paradigm in computer market has provided unprecedented computational capabilities. This also marks the end of the free-ride era—scientific software must now evolve with new chips. Hence, it is of great importance to develop large legacy-code optimization frameworks to achieve an optimal system architecture-algorithm mapping that maximizes processor utilization and thereby achieves higher application performance.

To address this challenge, this thesis studies and develops scalable algorithms for leveraging many-core resources optimally to improve the performance of massively parallel scientific applications. This work presents a systematic approach to optimize scientific codes on emerging architectures, which consists of three major steps: (1) Develop a performance profiling framework to identify application performance bottlenecks on clusters of emerging architectures; (2) explore common algorithmic kernels in a suite of real world scientific applications and develop performance tuning strategies to provide insight into how to maximally utilize underlying hardware; and (3) unify experience in performance optimization to develop a top-down optimization framework for the optimization of scientific applications on emerging high-performance computing platforms.

This thesis makes the following contributions. First, we have designed and implemented a performance analysis methodology for Cell-accelerated clusters. Two parallel scientific applications—lattice Boltzmann (LB) flow simulation and atomistic molecular dynamics (MD) simulation—are analyzed and valuable performance insights are gained on a Cell processor based PlayStation3 cluster as well as a hybrid Opteron+Cell based cluster similar to the design of Roadrunner—the first petaflop supercomputer of the world. Second, we have developed a novel parallelization framework for finite-difference time-domain applications. The approach is validated in a seismic-wave propagation simulation code on BlueGene/L, BlueGene/P and x86 quad-core processor based clusters. In addition, we have developed strategies for in-core optimization of the algorithmic kernel of this application, which is a high-order stencil computation—a common kernel to a spectrum of finite-differences based applications. Third, we have applied this systematic approach to a production level first-principles molecular-dynamics application, which has achieved a record of 2.58×10^{12} electronic degrees of freedom on 163,840 BlueGene/P processors. Finally, we have devised a systematic end-to-end performance optimization scheme for large-scale scientific applications on emerging high-performance computing platforms.

Chapter 1

Introduction

1.1 Significance of the Research

Computer simulation is the third mode of scientific research that bridges the gap between analytical theory and laboratory experiment. Experiments search for patterns in complex natural phenomena. Theories encode the discovered patterns into mathematical equations that provide predictive laws for the behavior of nature. Computer simulations solve these equations numerically in their full complexity, where analytical solutions are prohibitive due to a large number of degrees of freedom, nonlinearity, or lack of symmetry. In computer simulations, environments can be controlled with any desired accuracy and extreme conditions are accessible far beyond the scope of laboratory experiments.

This extraordinary new tool in the hands of the scientists, the computer simulation, is tightly connected to the exponential increase in the power of computers on which the computations are carried out in parallel. In order to meet the increasing performance requirements of parallel applications and to follow Moore's Law, chipmakers such as Intel, IBM, and AMD have shifted their production line towards multicore chips. Homogenous collection of such architectural innovations are being used in parallel systems such as the Intel Xeon and AMD Opteron clusters at the High Performance Computing and Communications (HPCC) [44] facility of the University of Southern California (USC). Furthermore, hybrid systems combining conventional general-purpose central processing units (CPUs) with specialized processors such as Cell BE processors, e.g., Roadrunner [7], and also recently graphical processing units (GPUs), e.g., Tianhe-1A [98], are among the fastest supercomputers of the world [99]. Achieving high application performance on today's complex petascale (10^{15} calculations per second) systems containing more than a hundred thousand computing nodes that include variety of accelerators is tightly coupled with designing an optimal architecture-algorithm mapping framework. Otherwise, while the peak performance of parallel machines increases, the gap between the peak and actual performance of the codes becomes wider.

1.2 Motivation and Challenges

The shift to multicore architecture design has provided unprecedented floating-point performance on a single chip, however often extra cores in such architectures go underutilized on modern systems. Also, it is a challenge to efficiently program on distributed systems featuring different flavors of memory hierarchy, CPUs or high performance interconnects produced by different vendors.

The recent advances also mark the end of the free-ride era—scientific software must now evolve with new chips. Consequently, legacy software needs to be systematically and efficiently ported to new hardware. Our goal is to allow scientists to maximally exploit the computational potential of high-end parallel machines to tackle larger problems with finer resolution. For this purpose, our research [23-26, 72, 77, 78] focuses on a variety of application contexts yet strives towards a general and flexible framework for systematic performance optimization on large-scale supercomputing platforms.

1.3 Background

In this section, we provide background information on our research. First, we introduce the use of performance monitoring at system and on-chip levels for detection of performance bottlenecks. Second, we discuss our multilevel optimization scheme for a representative scientific application without loss of generality. Third, we consider our metascalable computing framework. Forth, we provide background information on our production-level density functional theory based simulation.

1.3.1 Performance Monitoring

Application developers at the forefront of high-performance computing (HPC) have been investigating the use of hybrid architectures to improve application performance. Hybrid architectures attempt to improve application performance by combining conventional, general-purpose CPUs with any of a variety of more specialized processors such as GPUs, FPGAs, and Cells. The complexity stemming from hybrid architectures make understanding and reasoning about application performance difficult without appropriate tool support. An example to the recent hybrid architecture based supercomputers and the focus of our profiling efforts is Cell-based clusters, such as petascale Roadrunner supercomputer [7] (comprising 6,120 dual-core Opterons plus 12,240 PowerXCell 8i processors) in Los Alamos National Laboratory.

The Cell processor's complex architecture—eight *synergistic processing elements* (SPEs) managed by a single *power processor element* (PPE)—makes profiling tools essential for performance optimization. Traditional tools merely monitor performance events on PPEs, which provide less than 6% of PowerXCell 8i flops performance and are usually used for solely controlling SPE processes instead of computing. The IBM Cell Software Development Kit (SDK) [46] includes a Cell performance-debugging tool (PDT) that helps analyze the performance of a single Cell board (up to two Cell processors) with two PPEs that share the main memory, run under the same Linux operating system, and share up to 16 SPEs. PDT can trace only a specific set of SDK library functions such as SPE activation, DMA transfers, synchronization, signaling, and user-defined events. Because PDT involves the slow PPE on the critical path of tracing, the PPE can easily become a performance bottleneck and may even influence application performance. Another tool for analyzing Cell performance is Vampir [10], which Nagel et al. used to visualize intra-Cell events such as mailbox communication and DMA transfers [41].

1.3.2 Multilevel Optimization

As hierarchical multicore processors with complex computational and memory organizations emerge as a result of the quest for simultaneous performance and power-efficiency improvement, a challenge faced by software developers and application scientists is the adaptation of algorithms to effectively utilize this underlying hardware for broad computational applications. The emerging multicore paradigm has given us unprecedented supercomputing power [7], such as IBM BlueGene L and P, Roadrunner, and Cray Jaguar, through scaling at multiple levels, and in particular, multiple cores per node, interconnected into hierarchical systems of up to more than 100,000 cores. All have complex memory hierarchies, where some memory is shared across cores, some is dedicated, and some requires explicit management in software. While at different scales, the features of these architectures are mirrored in commercial microprocessors, which represent their constituent nodes, and are often combined into clusters of nodes as targets of high-end applications. From an application programmer's perspective, we hypothesize that such architectures can be viewed as hierarchical computational units with corresponding hierarchical storage that is explicitly or implicitly managed by software. The computation hierarchy includes support for fine-grain data-parallelism, through SIMD multimedia extensions such as Streaming SIMD Extensions 3 (SSE3) for Intel and AMD platforms and Altivec for PowerPC, through the SPEs of the IBM Cell, or through the streaming processors of an NVIDIA GTX 280. Across cores, thread-level parallelism permits potentially independent computation on related data, while across nodes, coarse-grain parallelism on independent data can be exploited. Data locality is critical to achieving high performance on such architectures, so memory structures including registers, multilevel caches and storage buffers should be carefully managed to match the hierarchical parallel constructs.

As a common computational kernel in a variety of scientific and engineering applications [52, 67], stencil computation (SC) has extensively been studied. For example, Datta et al. have used both cache-aware and cache-oblivious approaches to perform comprehensive SC optimization and auto-tuning on a variety of state-of-the-art architectures, including NVIDIA GTX280 [19]. Williams et al. [103] have optimized a lattice Boltzmann application on leading multicore platforms, including Intel Itanium2, Sun Niagara2, and STI Cell. Other approaches to SC optimization include tiling [84] and iteration skewing [32, 83, 104]. However, there has been little research on performance optimization of high-order stencil computations (HOSC), which are characterized by a large memory footprint of each stencil, spanning multiple levels of parallelization ranging from data to inter-core to inter-node levels. The pivotal role of HOSC in broad applications and the emergence of a wide landscape of heterogeneous multicore architectures have motivated us to develop a unified parallelization strategy that scales on massively parallel multicore supercomputers and perform systematic performance optimization on each of its hierarchical levels.

1.3.3 Metascalable Computing

The ever-increasing capability of high-end computing platforms is enabling unprecedented scales of first-principles based simulations to predict system-level behavior of complex systems [27]. An example is large-scale molecular-dynamics (MD) simulations involving multibillion atoms [63]. Such simulations can couple chemical reactions at the atomistic scale and mechanical processes at the mesoscopic scale to solve broad mechano-chemistry problems such as nanoenergetic reactions, in which reactive nanojets catalyze chemical reactions that do not occur otherwise [70]. A hard problem is to predict long-time dynamics, because the sequential bottleneck of time precludes efficient parallelization [79, 88].

The hardware environment is becoming challenging as well. Emerging sustained petaflops computers involve multicore processors [2], while the computer industry is facing a historical shift, in which Moore's law due to ever increasing clock speeds has been subsumed by increasing numbers of cores in microchips [22]. Thus scientific application programmers need to develop reusable "design once, scale on new architectures" (or metascalable) applications.

1.3.4 Density Functional Theory Method

There is growing interest in large-scale MD simulations involving multimillion atoms [1, 33, 66], in which interatomic forces are computed quantum mechanically [12, 29] in the framework of the density functional theory (DFT) [43, 50, 51] to accurately describe chemical reactions. Such large DFT-based MD simulations would provide requisite coupling of chemical reactions, atomistic processes, and long-range stress phenomena for broad applications. Examples are energetic materials, in which chemical reactions sustain shock waves, and stress corrosion, where chemical reactions at the crack tip need to be coupled with long-range stress fields. Unfortunately, DFT-based MD simulations are rarely performed over $N \sim 10^2$ atoms because of their $O(N^3)$ computational complexity, which severely limits their applicability.

Our computational approach toward density functional theory calculations is to perform a number of small DFT calculations "on the fly" to compute interatomic forces quantum mechanically during an MD simulation. The concurrent DFT-based MD approach is realized using an EDC density functional theory (EDC-DFT) algorithm [90, 105]. The DFT reduces the exponential complexity to O(N), by solving N_{el} one-electron problems self-consistently instead of one N_{el}-electron problem (the number of electrons, N_{el}, is on the order of N). The DFT problem can be formulated as the minimization of an energy functional with respect to electronic wave functions. In the EDC-DFT algorithm, the physical space is a union of overlapping domains, $\Omega = \cup_{\alpha} \Omega_{\alpha}$, and physical properties are computed as linear combinations of domain properties that in turn are computed from local electronic wave functions. For example, the electronic density $\rho(\mathbf{r})$ is calculated as $\rho(\mathbf{r}) = \Sigma_{\alpha}\, p^{\alpha}(\mathbf{r})\, \Sigma_{n}\, f(\varepsilon_{n}^{\alpha}) \,|\, \psi_{n}^{\alpha}(\mathbf{r}) \,|^{2}$, where the support function $p^{\alpha}(\mathbf{r})$ vanishes outside domain Ω_{α} and satisfies the sum rule, $\Sigma_{\alpha}\, p^{\alpha}(\mathbf{r}) = 1$, and $f(\varepsilon_{n}^{\alpha})$ is the Fermi distribution function corresponding to the energy ε_{n}^{α} of the n-th electronic wave function (or Kohn-Sham orbital) $\psi_{n}^{\alpha}(\mathbf{r})$ in Ω_{α}. For DFT calculation within each domain, we use a real-space approach based on high-order finite differencing [14], where iterative solutions are accelerated using the multigrid preconditioning [30]. The multigrid is augmented with high-resolution grids that are adaptively generated near the atoms to accurately operate atomic pseudopotentials [90]. The numerical core of EDC-DFT thus represents an HOSC [19, 25, 26].

1.4 Problem Statement

Today, HPC platforms depend on diverse processor architectures to meet computational demands of scientific simulations. Energy-efficient load-store architectures (e.g., PowerPC), complex instruction set pipelining processors (e.g., Intel Xeon, AMD Opteron), and computational throughput optimized accelerators (e.g., IBM Cell BE) provide bulk of the computational power for most supercomputers. To realize new levels of computing performance, the application developers must adapt their software to rapidly increasing heterogeneity and scale of new generation computing platforms. It is for this reason that we believe developing parallelization and optimization methodologies is a critical challenge to sustain exponential speed up of legacy software.

This dissertation research addresses the problem of developing a systematic end-to-end parallelization and optimization framework for high performance computer simulations on emerging computing platforms. Specifically, we will:

1. Develop productivity tools at system level to aid program optimization. In this context, we will focus on devising a memory efficient dynamic program analysis methodology for the new generation of hybrid computing clusters.
2. Design a parallelization framework that captures the organizational and architectural properties of the HPC platforms at the application level while enabling sustained scalability on emerging parallel computing systems.

3. Devise a systematic optimization scheme for broad scientific applications and verify the effectiveness for a production-level scientific code on available HPC architectures.

1.5 Contributions

So as to profile application performance on Cell-based clusters, we have developed a profiling method [23, 24]. We employ a *reverse acceleration* programming model in which the hybrid cluster architecture is presented to the programmer as a logical cluster of Cell SPE processors by using the Cell Messaging Layer (CML) [74]. CML provides a subset of the functions and semantics of the MPI standard [93] including point-to-point communication, broadcasts, barriers, and global reductions. Therefore it constitutes a key insertion point for profiler events.

Our approach can trace not only intra-Cell direct memory access (DMA) events but also inter-Cell message passing. Our implementation is efficient in terms of resource consumption (only 12 KiB of SPE local store memory is required) and has an overhead of less than 0.3 μs per profiler call for a typical scientific application executing on the Cell BE.

The key difference between our profiler [23, 24] and the earlier research mentioned in section 1.3.1 which targets profiling on Cell processor is that we perform *cluster-level* analysis for MPI programs running on compute nodes featuring a hybrid architecture comprising AMD Opterons/PowerXCell 8i processors such as Roadrunner and PlayStation3 (PS3) commercial gaming consoles featuring Cell BE processors. The underlying message-passing model of CML, which treats an entire cluster of Opterons+Cells (or PS3s) as a homogenous collection of SPEs, has a central importance to our cluster-wide analysis. In addition to monitoring the same types of intra-Cell events as existing Cell profilers, our implementation can log inter-Cell, inter-blade, and inter-node communication. We have tested our implementation on up to 256 SPEs, although there is nothing limiting us from scaling up to thousands or even tens of thousands of SPEs. To test the use and quantify the overhead of our profiler, we have ported two parallel scientific applications—lattice Boltzmann (LB) flow simulation and atomistic molecular dynamics (MD) simulation—to the PS3 and hybrid Opteron+Cell Roadrunner architecture using CML. We have also demonstrated two sample uses of the profiler (1) communication analysis; and (2) call-stack analysis.

We have developed a hierarchical scalable parallelization scheme for HOSC [25, 26] that features in-core level optimization techniques to exploit the floating-point performance of the computational units through efficient use of the hierarchical memory levels in modern multicore processors. Our multilevel approach combines: (1) data locality optimizations through auto-tuned tiling for efficient use of hierarchical memory; (2) register blocking and data parallelism via single-instruction multiple-data (SIMD) techniques to utilize registers and exploit data locality; (3) software prefetching to hide memory latency; (4) inter-core parallelization via multithreading; and (5) inter-node parallelization via spatial decomposition. We have illustrated the hierarchical scalable parallelization scheme by applying it to a 6th-order stencil based seismic wave propagation application. Our intra-node optimization using multithreading and SIMD parallelization has achieved a speedup of 5.83 for 8 threads on a single quadcore Intel Nehalem node. Our approach has reduced last level cache miss rate by 7.7×, and achieved 55% of the theoretical peak performance of at a single core of Intel Nehalem by incorporating loop tiling for Translation Lookaside Buffer (TLB), caches and registers, and explicit use of SSE instructions. We have obtained good overall strong scalability on all platforms, with even superlinear speedups on the Intel architecture. Excellent weak-scalability has been also achieved on the 256 processor Clovertown-based cluster and 32,768 processors of BlueGene/P.

To address metascalability challenge, we have developed key technologies for parallel computing with portable scalability [72]. These include an embedded divide-and-conquer (EDC) algorithmic framework to design linear-scaling algorithms for broad scientific and engineering applications (e.g. equation solvers, constrained optimization, search, visualization, and graphs involving massive data) based on spatial locality principles [62]. This, combined with a tunable hierarchical cellular decomposition (HCD) parallelization framework, maximally exposes concurrency and data locality, thereby achieving reusable "design once, scale on new architectures" (or metascalable) applications. It is expected that such metascalable algorithms will continue to scale on future multicore architectures. The "seven dwarfs" (a dwarf is an algorithmic method that captures a pattern of computation and communication), which were first identified by Phillip Colella, have been used widely to develop scalable parallel programming models and architectures [2]. We expect that the EDC-HCD framework will serve as a “metascalable dwarf” to represent broad large-scale scientific and engineering applications.

We have applied the EDC-HCD framework to a hierarchy of atomistic simulation methods. In MD simulation, the system is represented by a set of *N* point atoms whose trajectories are followed to study material properties [34, 65, 79]. Quantum mechanical (QM) simulation further treats electronic wave functions explicitly to describe chemical reactions [40, 68, 90]. To seamlessly couple MD and QM simulations, we have found it beneficial to introduce an intermediate layer, a first principles-based reactive force field (ReaxFF) approach [69, 100], in which interatomic interaction adapts dynamically to the local environment to describe chemical reactions. The ReaxFF is trained by performing thousands of small QM calculations.

Finally, we have unified our experience in performance optimization to develop a systematic end-to-end performance optimization scheme for large-scale scientific applications on emerging high-performance computing platforms. We expect that the optimization of production level scientific codes on high-end machines, such as our first principles MD application on BlueGene/P, can potentially save one-to-two orders of magnitude of petaflops•days of computing resources for a typical run or can extend the time and length scale of real-life MD simulations at the same computational cost, and contribute in scientific discovery through computer computation.

1.6 Structure of This Thesis

The rest of this text is organized as follows. An MPI performance-monitoring interface is examined in Chapter 2. A multilevel optimization scheme for HOSC is detailed in Chapter 3. A metascalable computing framework is discussed in Chapter 4. Our systematic approach to optimizing a production-level density functional theory code is described in Chapter 5. We summarize our conclusions and discuss the direction of the future work in Chapter 6.

Chapter 2

MPI Performance Monitoring Interface

In this chapter, we present a methodology for profiling parallel applications executing on the family of architectures commonly referred as the “Cell” processor. Specifically, we examine Cell-centric MPI programs on hybrid clusters containing multiple Opteron and IBM PowerXCell 8i processors per node such as those used in the petascale Roadrunner system. We analyze the performance of our approach on a PlayStation3 console based on Cell Broadband Engine—the CBE—as well as an IBM BladeCenter QS22 based on PowerXCell 8i. Our implementation incurs less than 0.5% overhead and 0.3 μs per profiler call for a typical molecular dynamics code on the Cell BE while efficiently utilizing the limited local store of the Cell’s SPE cores. Our worst-case overhead analysis on the PowerXCell 8i costs 3.2 μs per profiler call while using only two 5 KiB buffers. We demonstrate the use of our profiler on a cluster of hybrid nodes running a suite of scientific applications. Our analyses of inter-SPE communication (across the entire cluster) and function call patterns provide valuable information that can be used to optimize application performance.

2.1 Architectural Background and Testbed

In this section, we describe the architecture of the Cell Broadband Engine and PowerXCell 8i that provides the bulk of the performance of our target cluster and the focus of our profiler study. We then briefly summarize the overall architecture of our testbed. Finally, we describe the Cell Messaging Layer, which is an enabling technology for exploiting hybrid (or completely cell based) clusters and therefore a key insertion point for profiler events.

2.1.1 Cell Broadband Engine and IBM PowerXCell 8i

Cell BE has a heterogeneous architecture incorporating a power processor element (PPE) and eight synergetic processing elements (SPEs) on the same chip. SPEs are connected via an element interconnect bus (EIB), which supports a peak bandwidth of 204.8 GB/s for intra-chip data transfers among the PPE, SPEs, the memory, and the I/O interface controllers [16]. A single Cell BE has a peak single-precision performance of 217.6 Gflops/s for which it took attention of the high performance computing community in the recent years [37], whereas its double-precision peak is limited to 21 Gflops/s.

The IBM PowerXCell 8i (also referred as the Cell extended Double-Precision, Cell-eDP) is the latest implementation of the Cell BE featuring 108.8 Gflops/s on double-precision operations. It drives one of the fastest supercomputers at the time of this writing, Roadrunner at Los Alamos [7]. Each SPE of PowerXCell 8i contains a 3.2 GHz synergetic processing unit (SPU) core, 256 KB of a private, program-managed local store (LS) in place of a cache, and a memory flow controller (MFC) that provides DMA access to main memory. The SPE uses its LS for efficient instruction and data access, but it also has full access (via DMA) to the coherent shared memory, including the memory-mapped I/O space.

To make efficient use of the EIB and to interleave computation and data transfer, the PPE and 8 SPEs are equipped with a DMA engine. Since an SPE's load/store instructions can access only its private LS, each SPE depends exclusively on DMA operations to transfer data to and from the main memory and other SPEs' local memories. The use of DMAs as a central means of intra-chip data transfer maximizes asynchrony and concurrency in data processing inside a Cell processor [38].

2.1.2 Testbed

The PS3 features an identical Cell BE to the ones in IBM BladeCenter QS20. Recently the gaming console has been used as a low-cost computing platform by scientists [71]. However, one of the SPEs is disabled in PS3s for chip yield reasons and another SPE is reserved for use by GameOS operating system that acts as a hypervisor, and virtualizes the system resources. Out of 256 MB Rambus Extreme Data Rate (XDR) memory on PS3, only 200 MB is accessible to Linux OS and applications. Even though PS3s are not crafted for high performance cluster computing [11], they offer a valuable testing platform for tools targeting Cell based architectures. In our study we use a PS3 console to quantify the overhead that our profiling library incurs.

Our second testing platform comprises 8 nodes, called tri-blades, where each tri-blade has two IBM QS22 Cell blades and one IBM LS21 AMD Opteron blade. The QS22 contains two PowerXCell 8i processors running at 3.2 GHz and each with an associated 4 GB of DDR2 memory. The LS21 blade includes two dual-core Opteron cores clocked at 1.8 GHz. Each tri-blade has a single connection to a Mellanox 4x DDR InfiniBand network. Typically, the Opterons handle mundane processing (e.g., file system I/O) while mathematically intensive elements are directed to the Cell processors. Each tri-blade in our testbed is architecturally identical to the tri-blades used in Roadrunner.

2.1.3 Cell Messaging Layer

CML is an implementation of common MPI functions for SPE-to-SPE communication in Cell-based clusters. The programming model underlying the CML is that applications run entirely on the SPEs. The SPE-centric model of CML assigns unique MPI ranks to each SPE assigned to an application. By means of using PPE (and possibly conventional CPUs like Opterons if they exist in the cluster) primarily for shuttling messages to SPEs in other blades (or PS3s) instead of computation, the abstraction provided by CML allows each SPE to communicate with other SPEs regardless of whether the SPEs are in the same socket, the same blade, the same node, or different nodes. On a cluster of Cells, CML implements a mechanism for forwarding data from a SPE to its PPE then across a network to a remote PPE and finally to the target SPE. The PPE needs to be involved because a SPE cannot interact directly with I/O-bus devices such as network interface cards (NICs). In addition to handling communication operations, PPE, also initializes CML, starts SPE programs and waits until all SPEs invoke MPI_Finalize(), and finally shuts down the CML. Therefore both SPE/PPE programs need definitions of CML functions and should be linked with CML libraries, whereas SPE program can run an existing MPI application with only minor modifications that are necessary due to architectural requirements of the Cell. In effect, we have ported our scientific applications relatively easily to both of our testing platforms.

CML also provides Programmer's Message Passing Interface (PMPI) functions [56] which have a one-to-one correspondence to MPI calls. This interface enables any calls made to the MPI functions, by the SPEs, to be intercepted and thus recorded. Section 2.2.2 discusses the use of PMPI calls within our profiler.

CML also offers a remote procedure call (RPC) mechanism through which SPEs can invoke a function on the PPE (PPEs can subsequently call a function on the accompanying host CPU if it exists) and receive any results. This capability is particularly useful for our profiler, where local SPEs need to call a PPE malloc() to allocate space in PPE memory to hold the entire list of recorded events.

2.2 Software Design Details

Our implementation of the tracing library targets clusters of Cell processors. Each PPE within a Cell processor is responsible for synchronizing the program run on its SPEs. CML enables the total number of SPEs, as seen by an application, to scale: from a single processor containing eight SPEs to clusters of PS3s [71], or to Roadrunner that contains 97,920 SPE cores. The remainder of this section outlines the design and implementation of the profiler including its memory use, and events that are profiled.

2.2.1 Data Structures

The buffers that are used in the profiler implementation, along with the double-buffering operation of the buffers in the LS, are shown in Fig. 2.1. This is discussed further in section 2.2.2.

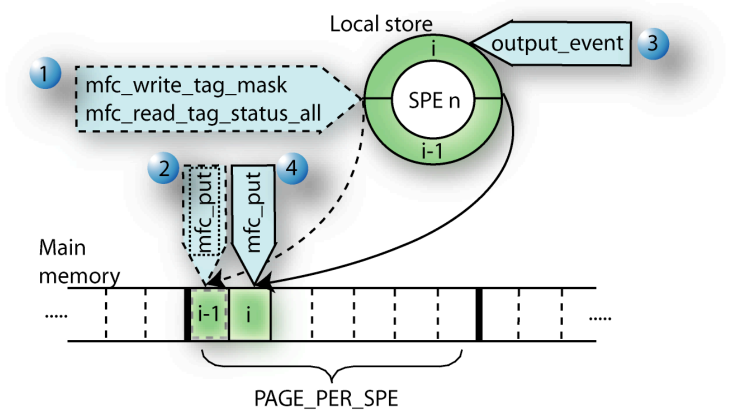


Figure 2.1: Operation of the double-buffering design

A cyclical pattern is used in Fig. 2.1 to illustrate the allocation of buffers in LS. They switch roles repeatedly—while one is being used to record newly created events, the other is being dumped to PPE main memory. In comparison, the PPE memory layout is linear, where each small section, or event page, corresponds to the size of a single buffer in LS.

Table 2.1 summarizes the structure of profile events and of event pages that hold a number of events. It is crucial that the events and the buffers are allocated to fit the 16-byte boundary required for DMA transactions. *ALIGNED16* is a short-hand notation for the *__attribute__((aligned (16)))* attribute, which specifies to the compiler to allocate the data structure to be 16 byte aligned. It is also important that 16 byte aligned profile data is structured the same both on SPE and PPE memories.

Table 2.1: Definitions of the data structures

```
#define PAGE_SIZE 64
#define ALIGNED16
__attribute__((aligned (16)))

  typedef enum    { PROFILE_START,
PROFILE_STOP, E, X, MPI_SEND,
MPI_RECV, MPI_ALLREDUCE, MPI_REDUCE,
MPI_BARRIER, MPI_BCAST, MFC_PUT,
MFC_PUT64, MFC_GET64, MFC_PUT32,
MFC_GET32} event_type_t;

typedef struct    {
     double time_stamp;
double duration;
event_type_t type;
unsigned long long enx;
unsigned long long exx;
short output_flag;
int data[8] ALIGNED16;
       }ALIGNED16 event_record_t;

typedef struct page_tag {
struct page_tag* next_page;
event_record_t events[PAGE_SIZE];
       }ALIGNED16 event_page_t;
```

The enumerator lists the type of events our implementation can currently monitor. We record calls to the profiler start/stop functions, SPE function entry/exit (*E, X*), calls to the MPI functions implemented in CML (*MPI_SEND, MPI_RECV, MPI_ALLREDUCE, MPI_REDUCE, MPI_BARRIER, MPI_BCAST*) and various DMA put/get transactions which are issued by functions *spu_mfcdma32*() and *spu_mfcdma64*() defined in *libspe2*—

the standard SPE library included in the IBM Cell/B.E. SDK. We have limited our implementation to cover only relevant DMA transaction types to our test applications.

In addition to recording the type of event, *event_record_t* also records a time stamp and the duration of an event, address of the called SPE function and its caller (*enx ,exx*), an output flag to indicate that an event has happened and *data* array which includes destination/source, send/receive size and send/receive counts for MPI events. Effective addresses (*enx, exx*) are stored as an *unsigned long long* on both the SPE and PPE, so that they can be treated in a unified fashion no matter if the PPE code is compiled for 32-bit or 64-bit execution. One event record uses 80 bytes in memory.

A single buffer, or event page, is defined by *event_page_t*. The size of a page was set to be 64 in our testing (using 5,120 bytes). A pointer to the next page to use is a part of *event_page_t* in case the current page fills up. Contrary to the SPE, which has two event pages, the PPE allocates a far greater number of event pages. For our analysis on the PowerXCell 8i based hybrid cluster, the PPE allocates 10,000 event pages per SPE giving a total PPE memory footprint of 400 MiB (=8×10,000×64×80). However, in our PS3 benchmarks we had to limit the PPE memory allocation to less than 200 MB due to 20 times less PPE addressable memory in PS3. In fact, we have observed that as long as enough PPE memory is reserved, the performance of the profiler is not affected.

2.2.2 Implementation

CML based applications first start on the PPEs, which subsequently launch code on the SPEs. When the profiler is enabled, an instrumented SPE program, once launched, immediately invokes an allocation function on the PPE, using the CML's RPC mechanism, for event pages in main memory. Each SPE is returned the base address of the reserved memory via the same RPC mechanism. Before a SPE proceeds with actual application execution, it allocates two event buffers in its LS. However, this allocation is much smaller than its counterpart in main memory due to the limited size of the LS. In our tests the profiler statically allocates only two small event buffers of size 5,120 bytes, which holds up to 64 events, in SPE memory. Apart from the 10 KiB required for the two buffers, the profiler code requires less than an additional 2 KiB in LS but is dependent on the actual number of CML functions used by an application. This is ~30,000 times smaller than the memory used by our profiler in the main memory of the PPE of a PowerXCell 8i.

Profiler initialization is followed by the execution of the actual MPI application. Throughout the application run, the instrumented functions are called to record events. The instrumented operations, as provided by the profiler, create event logs. For instance, an SPE-to-SPE message-passing request invokes the corresponding instrumented MPI communication operation, which populates the event data structure with the relevant information, e.g., type, source/target, size of the message, and secondly calls the corresponding PMPI routine, which is implemented by CML, to send the actual message. The profiler library provides similar instrumented functions to profile other events including DMA operations and SPE function call activities.

SPE LS memory is limited to 256 KB. If it were to be filled with trace data, it would inhibit the execution of the SPE code. In order to circumvent this possibility, we use a double-buffering approach [86] to log trace events. Instead of continuously pushing events to a dynamically increasing allocation in LS, SPU writes profile event logs as they appear to one of the two small buffers allocated during profiler initialization. Once the buffer being used is full, previous buffer-dump operation is checked for completion (Step 1 in Fig. 2.1), by using *mfc_write_tag_mask* and *mfc_read_tag_status_al,* in order to avoid overwriting data being transferred. If the preceding dump has been completed, a non-blocking DMA (*mfc_put*) is issued to transfer the buffer to main memory (Step 2 in Fig. 2.1). Each SPE sends the data to a privately reserved address, which it determines by using the memory base address received through the RPC mechanism during initialization, its local rank and number of previous dumps it has performed up to then. The SPE also switches the trace buffers and uses the available buffer to record new events (Step 3 in Fig. 2.1). Meanwhile, the SPE execution continues without interruption as a non-blocking DMA is used. Once the second buffer is filled, the SPE switches buffers again and continues with recording events to the first buffer as it issues a DMA transfer (*mfc_put*) to dump the second buffer to the end of the preceding dump in the main memory (Step 4 in Fig. 2.1). If the speed of event generation is faster than the time taken to transfer a single LS buffer to main memory then the application execution will pause. In such a case the size of the LS buffers can be increased but clearly at a reduction in the size of the LS store available to the application.

The double-buffering implementation not only overlaps data dumping with program execution, but also gives the capability of logging in excess of 10^4 times more events than the LS could have stored by using just two small buffers, and leaves more LS available for program and data in each SPE.

Upon the termination of tracing, the SPE program dumps the last buffer, regardless of how full it is, to main memory. Once all of the SPEs terminate the PPE writes the profile data from main memory to several files, one per SPE, which contains the events that are ordered in terms of their time of occurrence. The output files can be post-processed for numerous performance analysis studies.

2.3 Analysis

In this section, we first provide detailed analysis of the overhead incurred by the profiling activity on a single Cell BE processor of the PS3. Secondly, we compare the overhead on a single PowerXCell 8i to that on Cell BE and finally delineate cluster-wide use of the profiler.

Three applications were chosen to both quantify the overheads of the profiler use and also to illustrate its usefulness. The first application is Sweep3D, which solves a single-group time-dependent discrete ordinates neutron-transport problem. It processes a regular three-dimensional data grid, which is partitioned onto a logical two-dimensional processor array. Its computation consists of a succession of 3D wave fronts (sweeps), in which each processor receives boundary data from upstream neighbors, performs a computation on its local sub-grid, and produces boundaries for downstream neighbors. All communications use MPI to transfer boundary data to neighboring processors.

The second application is molecular-dynamics (MD) [63]. The MD simulation follows the time evolution of the positions of *N* atoms by solving coupled ordinary differential equations. For parallelization, the MD code uses a 3-D spatial domain that is partitioned in all three dimensions into *P* sub-grids of equal volume. Each step in the simulation requires the processing of the local sub-grid as well as boundary exchanges in each of 6 neighboring directions (i.e. the lower and higher neighbor subsystems in the x, y and z directions).

The third application is a lattice Boltzmann (LB) method for fluid flow simulations. The cellular-automata like application represents fluid by a density function on of the grid points on a regular 3D lattice [71]. LB exhibits the same 3D communication pattern as for MD where each time step involves DF updates and inter-sub-grid density migrations.

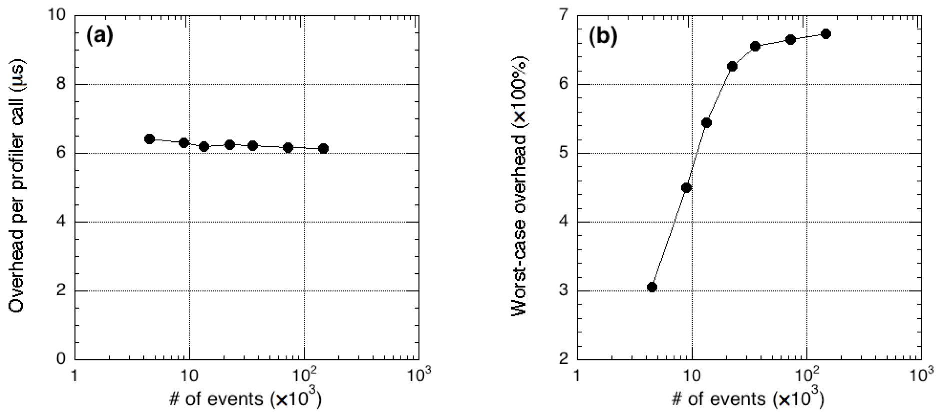
2.3.1 Performance Overhead Analysis

In effect, the performance overhead of the profiler is dependent on the application as the mixture of communication and computation operations vary from code to code. Therefore in this section we use an overhead metric by considering a worst-case scenario by using a kernel application containing only communication calls and no computation. Additionally, by executing the kernel on a single Cell processor we ensure that only fast on-chip communications over the EIB are used. The kernel application simply contains the communication pattern of the Sweep3D application thus resulting in a maximum rate of event generation. We provide the results first on Cell BE and second on PowerXCell 8i in the remainder of this section.

2.3.1.1 Cell BE

In order to quantify the profiling overhead we have performed a suite of tests on the Cell BE of the PS3, which represents typical node of our target cluster.

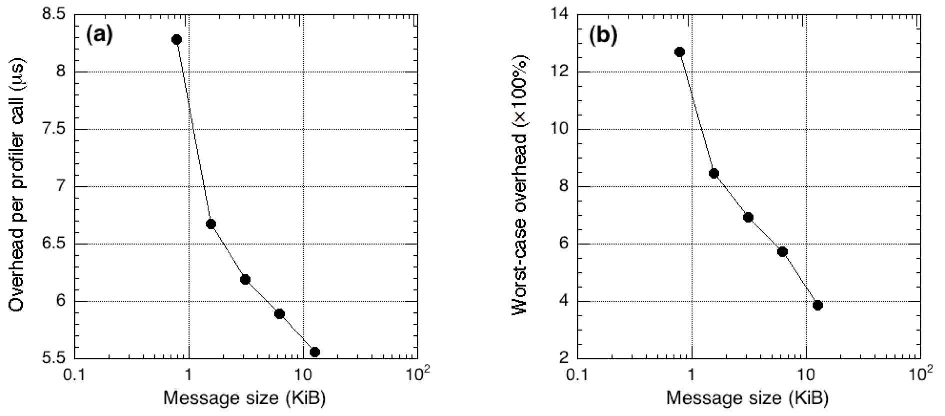
An equal number of MPI send and receive calls, using a fixed size of 600 doubles (4,800 bytes), for the 6 functional SPEs on the single Cell BE of the PS3 were used for the results shown in Fig. 2.2. Fig. 2.2(a) shows the average overhead for each profiler call as a function of the number of events and Fig. 2.2(b) shows the slowdown when varying the number of events (the x-axis shows the logarithm of the number of events). It can be seen that the average time required to record a single event is less than 6.3 μs. This corresponds to a slowdown of a factor of 6.8 for large numbers of events as shown in Fig. 2.2(b).



(a) Average cost per profiler call (b) Slowdown due to the profiler

Figure 2.2: Performance overheads of the profiler (6 SPE run on a single Cell BE)

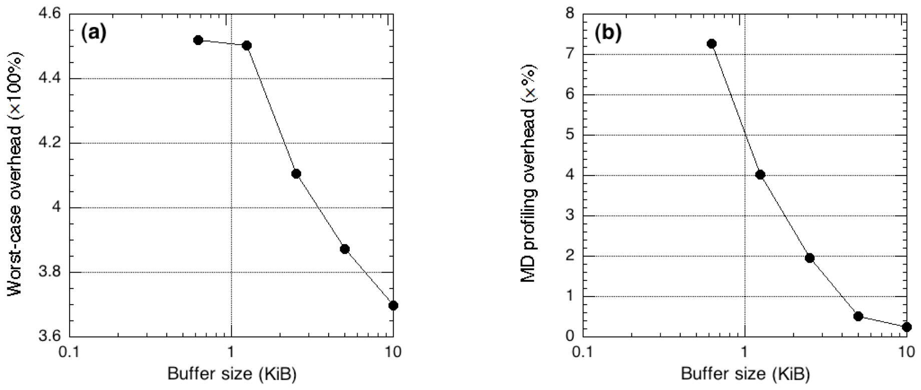
In order to quantify the effect of message size on profiling overhead, we have performed benchmarks on the Cell BE, and the results are shown in Fig. 2.3. Here, we fix the event count at 36,000, which is the point where saturation starts in Fig. 2.2(b), and keep the buffer size at 5 KiB while varying the sizes of the sent/received messages.



(a) Average cost per profiler call (b) Slowdown due to the profiler

Figure 2.3: Performance overheads of the profiler for varying message sizes (6 SPE run on a single Cell BE)

Fig. 2.3(a) shows that the smallest per profiler call overhead is less than 5.6 μs for 12.5 KiB sized messages, whereas Fig. 2.3(b) shows 4× slowdown factor. In comparison to Fig. 2.2, which used 4,800-bytes messages, profiler shows a better performance for 12.5 KiB messages. This can be attributed to the fact that larger transfers take longer time to complete, which is overlapped by keeping record of profile events, thereby reducing the profiler overhead.



(a) Slowdown in worst-case kernel (b) Slowdown in MD application

Figure 2.4: Performance overhead of the profiler for varying SPE buffer sizes on the Cell BE

In Fig. 2.4, we study the effect of changing the sizes of the double buffers reserved for profile events at SPEs of the Cell BE processor. In Fig. 2.4(a), we fix the event count at the saturation point of Fig. 2.2(b), i.e. 36,000, the message size at 12.5 KiB and vary the buffer size. It is observed that increasing the buffer size decreases the slowdown factor to as low as 3.7. As the buffer size at SPE increases, it takes less DMA transfers to PPE to dump the filled buffers, thereby increasing the performance of the profiler. However, it should be noted that there is a trade off between increasing the buffer size to achieve better profiler performance and the application performance itself because of the limited local store of SPEs. In selecting the buffer size, the memory requirements of application for its instructions and data should also be considered.

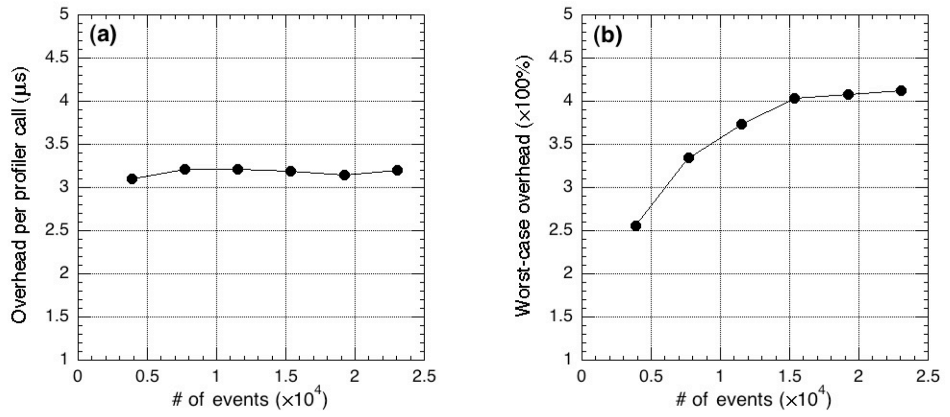
In order to quantify the profiling overhead for a typical scientific application, we port our parallel molecular-dynamics (MD) code [63] to PS3 using CML for handling MPI operations. Fig. 2.4(b) shows the overhead incurred by profiling of the message passing events of the MD application on a single PS3. The MD application implements a velocity-Verlet scheme, and calls several small functions at each time step for calculating atomic positions and velocities besides communication functions. Therefore, in Fig. 2.4(b), we turn off the function entry/exit tracing feature of the profiler in order to analyze MPI calls only, which are mainly for exchanging boundary-atom information. The results are the averages over a 100-time step simulation. Similar to Fig. 2.4(a), increasing the buffer size decreases profiling overhead. However, it should be noted that the y-axis of Fig. 2.4(b) is in percentages, i.e., profiler overhead is less than 0.5% for 10 KiB buffers. In effect, the cost of a single profiler call is less than 0.3 μs when 10 KiB buffers are used. The MPI messages in the benchmark shown in Fig. 2.4(b) are on-chip communications and use the EIB of the Cell processor, which is much faster in comparison to current high performance computing interconnects. Therefore in a cluster-wide analysis of an MPI-centric scientific application, the overhead incurred by logging MPI events will be extremely low.

2.3.1.2 PowerXCell 8i

Here, we compare the overhead of the profiler on PowerXCell 8i to that on Cell BE before we proceed with PowerXCell 8i based cluster-wide experiments in the next section.

The overhead evaluation benchmark we described in the experiment of Fig. 2.2 is repeated on 8 SPEs of a single PowerXCell 8i processor, and the results are shown in Fig. 2.5. Fig. 2.5(a) shows that the average time required to record a single event is less than

3.2 μs and slowdown factor rises up to 4.2 for larger numbers of events as shown in Fig. 2.5(b). Recall however that intra-cell communications take full advantage of the EIB which has a total bandwidth of 204.8 GB/s. Intra-cell communications using CML actually achieve a bandwidth of ~23 GB/s and latency of ~0.3 μs as shown in Table 2.2. And hence a SPE-to-SPE message of size 4,800 bytes takes less than 1 μs within a single Cell.



(a) Average cost per profiler call (b) Slowdown due to the profiler

Figure 2.5: Performance overheads of the profiler (8 SPE run on a single PowerXCell 8i)

It should be noted that the x-axis is linear in Fig. 2.5 as opposed to the logarithmic scale in Fig. 2.2. This is because the saturation of slowdown factor happens faster in comparison to Cell BE benchmark. The drop down in the average cost per profiler call and the slowdown factor in Fig. 2.5 in comparison to Fig. 2.2 can be explained by the fact that the Linux kernel in Cell BE of PS3 is running on top of a hypervisor which uses one SPE and also the EIB. On the PS3, the SPUs are hidden behind the hypervisor and every access happens in cooperation with the hypervisor. As a result writing to the local store or shuttling messages results in writing into kernel memory that represents the local store, which affects I/O of a given process and incurs additional overhead.

For a typical application running on a cluster of Cells, or on a hybrid processor configuration like Roadrunner, SPE-to-SPE communications can be significantly more costly than the ones considered in the worst-case discussed above. For instance, on Roadrunner, communications between SPEs on different nodes have a latency of over 11.7 μs at small message bandwidth of 161 MB/s. Therefore, in practice, the profiling overhead is much lower due to the increasing cost of communications as demonstrated in Fig. 2.4(b).

2.3.2 Cluster-wide Profiling

In this section we illustrate the usefulness of our profiling implementation for LB, MD and Sweep3D applications. We perform our tests on 8 nodes (32 SPEs per node) of a Roadrunner-like PowerXCell 8i based cluster.

2.3.2.1 Communication Analysis

The information that is generated by the profiler is analyzed off-line. One log-file is generated for each SPE used by the application. Fig. 2.6 shows an example of the SPE-to-SPE communication pattern of the original Sweep3D code.

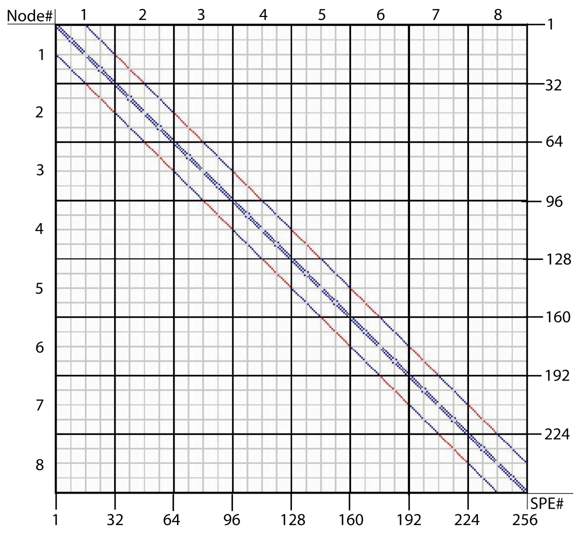


Figure 2.6: Communication pattern of the original Sweep3D

In Fig. 2.6, a larger square surrounded by thick lines and denoted by a node number, which contains 4×4 small squares, represents a tri-blade in the cluster. Each smaller square represents one Cell processor with 8 SPEs. The vertical and horizontal axes represent the sender and receiver SPE MPI ranks, and a colored pixel on the graph indicates a pair of communicating SPEs. The pixels are color coded to distinguish intra-node (blue) and inter-node (red) communications respectively.

The decomposition of Sweep3D's global grid onto a logical 2-D processor array can be seen in Fig. 2.6. Each processor communicates with its neighbor in the logical x and y directions. For a 256 processor run, the 2-D processor array consists of 16×16 SPE processors. Each processor communicates with its x neighbors (±1) as illustrated by the two sub-diagonals, and with its y neighbors (±16) indicated by the outermost two off-diagonals. Message passing for the two x neighbors is performed on the same chip through high-bandwidth (25.6 GB/s) EIB; whereas communication with the y neighbors corresponds to 1 message passing to another SPE residing within the same node but on the other Opteron and performed over PCIe via DaCS; and 1 message passing to an SPE on another node which adds InfiniBand in the path. These different inter-SPE communications incur different latency and bandwidth costs as shown in Table 2.2, which shows both the latency and bandwidth measured by ping-pong communication tests for CML.

Table 2.2: CML point-to-point performance

Configuration	Latency	Bandwidth
Same Cell	0.272 μs	22,994.2 MB/s
Same node	0.825 μs	4,281.3 MB/s
Different nodes	11.771 μs	161.2 MB/s

The high latency of inter-node communication in comparison to intra-cell communication stems from the involvement of PPEs and Opterons in the former. To achieve higher performance, parallel algorithms should be designed to exploit the low latency and high bandwidth of EIB connecting intra-cell SPEs and avoid inter-node communication wherever possible. Fig. 2.7 shows the communication pattern of a modified version of Sweep3D, which performs much of the message passing activity over the EIB.

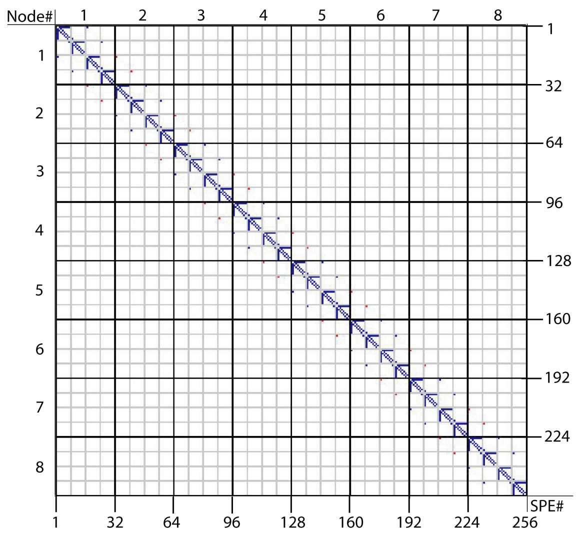


Figure 2.7: Communication pattern of the modified Sweep3D

In the modified version of Sweep3D, one SPE of each Cell acts as a root and exclusively handles inter-node message passing by gathering messages from the other SPEs on the same chip and sending it to the root on the destination Cell. This reduces the number of inter-cell messages significantly and promises an increase in performance. For comparison, Fig. 2.8 shows the communication pattern of MD for 256 SPE run.

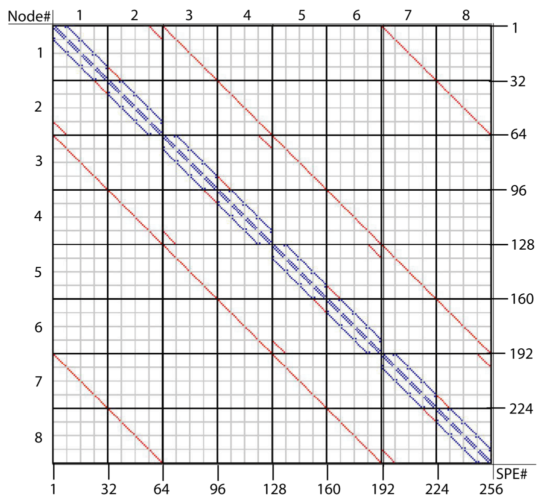


Figure 2.8: Communication pattern of MD

In Fig. 2.8, the logical arrangement of processors is in an 8×8×4 processor array. Each SPE performs two intra-cell communications with x neighbors. Communications to y neighbors is comparably slower with half of the SPEs requiring inter-node communications. For example, in the first node, SPEs 1–8 and SPEs 25–32 have one of their y neighbors in the next node, while for SPEs 9-24 the communications to y neighbors only involves intra-node communications. For all SPEs, message passing with z neighbors is inter-node communication with a high communication cost. This suggests a possible optimization, to increase the number of communications over the EIB, as with Sweep3D.

The volume of messages in MD is fairly regular. However, for applications where message send/receive activities and message sizes vary, heavier communication paths should be paid more attention. This is demonstrated with another application—LB in Fig 2.9.

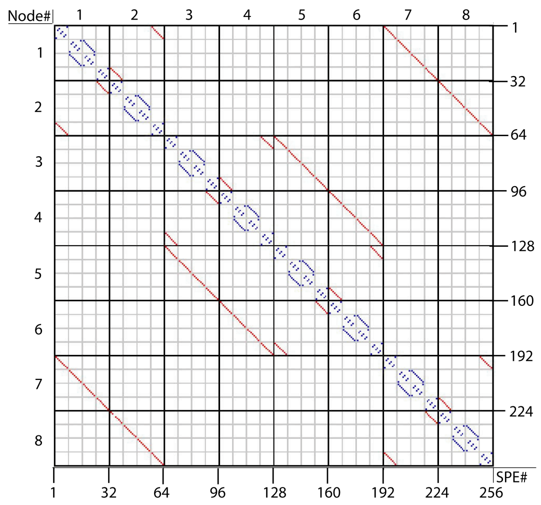


Figure 2.9: Heavy communications of LB

LB and MD have the same 3D communication pattern, however for LB, some message passing events are an order-of-magnitude smaller than the others. Therefore, in Fig. 2.9, instead of plotting all communications, we have drawn only heavier communications. It should be noted that Fig. 2.9 has equal numbers of blue and red dots, indicating that 50% of message passing activity is inter-node. The other half of the communication is intra-node, of which 1/3 happens among SPEs on different Cells of the node. Therefore, only 1/3 of heavier communication is taking advantage of the fast EIB. This suggests that LB suffers a larger communication cost in comparison with MD and that there is more room for optimization through the rearrangement of messages.

As a matter of fact, event data structure as described in section 2.2.1 has enough data to provide finer details on message passing events. For example inter-SPE and/or SPE-to-PPE communications can be analyzed in finer detail. Function use, duration, type of message passing activity, size of the message, type of data being sent (and/or received), count of a certain data type, and source/destination, can be analyzed to provide more insight into the program flow. As we can extract point-to-point communication matrix from an application execution, it is also possible to automatically identify the communication pattern by measuring the degree of match between point-to-point communication matrix and predefined communication templates for regularly occurring communication patterns in scientific applications [47].

2.3.2.2 Call-stack Analysis

The profiler library can also keep track of function entry and exits. This section illustrates the use of this functionality by a call-stack analysis as another use of our profiler.

Fig. 2.10 shows the function call graph for the execution at the first SPE of a 256-SPE run for LB code. Instrumentation for 10 iterations is visualized and only a portion of call graph is provided for the clarity of presentation. The node shown as the root is the main function, which calls collision, streaming and communication functions once during every iteration. The nodes for these 3 functions include the source file name and the source code line information, which are looked up from a symbol table during post-processing. The node, which calls the *MPI_SEND/MPI_RECEIVE* implementation of CML, represents calls to the communication functions. Its children nodes show source/destination and data count of the message in parenthesis. The edges of the graph are marked with the number of times a particular event is observed. For instance, the node *MPI_SEND(0,56,132)* represents SPE 0 sends a message to SPE 56 of 132 bytes and it has occurred 10 times during the profiling.

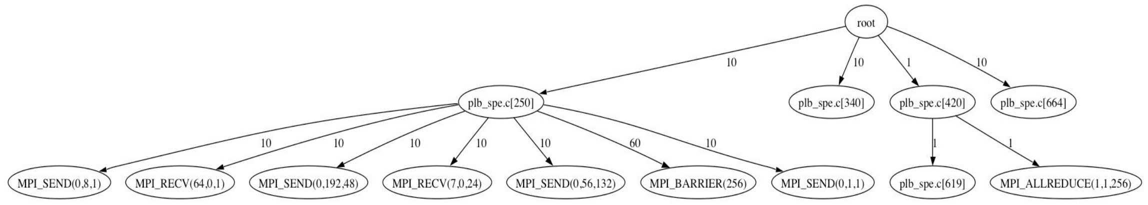


Figure 2.10: Function call graph for LB

The instrumentation is also done for functions expanded inline in other functions. The profiling calls indicate where the inline function is entered and exited. This requires that addressable versions of such functions must be available. A function may be given the attribute *no_instrument_function*, in which case this instrumentation will not be done. This can be used, for example, for high priority interrupt routines, and any function from which the profiling functions cannot safely be called, for example signal handlers.

The function call graph provides insight into program execution on a particular SPE on a cluster contributing to optimizations at the SPE level. In Fig. 2.10, we have weighted the edges with function call numbers. Instead, operation completion time could be used as an alternative for weighting as the profiler keeps durations of events as well. A call graph can be used to identify bottlenecks of performance at the SPE level and shed a light on required algorithm modifications for improvement.

2.4 Conclusion and Future Work

We have developed a low-memory-footprint (12 KiB of local store), minimally intrusive profiling library for parallel applications running on clusters of Cell processors. Our library overlaps computations and DMA transfers to reduce application perturbation and efficiently utilizes the small amount of SPE local store available on Cell processors.

We have analyzed the performance of our profiler on the Cell BE processor of a PlayStation3 and explored profiler performance for varying design and application specific parameters, such as buffer and message size. We have used our profiler library to analyze the performance of parallel scientific applications that run across multiple Cell processors, Cell blades, and cluster nodes. Inter-blade communication analysis for Sweep3D has shown how communication structure can affect application performance. We have ported two additional applications, LB and MD, to a hybrid Opteron+Cell cluster, and our profiler data suggests possible optimization opportunities. In order to demonstrate other uses of our library, we have analyzed the function-call pattern of a single SPE's program flow and used that to determine performance bottlenecks on the level of a SPE core.

While our study demonstrates high-speed, low-memory-overhead profiling for clusters augmented with Cell processors, it is certainly possible to optimize the profiler to further reduce its profiling cost and memory footprint. For example, the various types of profile events have different memory requirements (e.g., call-stack address records use only 16 bytes out of the 80 bytes allocated for the general event type). Therefore, restructuring the data types to be event-specific and adding compile-time options to customize the desired performance report may result in lower intrusion to program flow and reduce the post-processing effort for profile data.

Chapter 3

Multilevel Optimization of Stencil Computations

In this chapter, we present a scalable parallelization scheme for high-order stencil computations that also optimizes memory behavior on multicore clusters. Our multilevel approach combines: (1) data locality optimizations through auto-tuned tiling for efficient use of hierarchical memory; (2) register blocking and data parallelism via single-instruction multiple-data (SIMD) techniques to utilize registers and exploit data locality; (3) software prefetching to hide memory latency; (4) inter-core parallelization via multithreading; and (5) inter-node parallelization via spatial decomposition. The scheme is applied to a 6th-order stencil based seismic wave propagation code. Tiling optimizations achieve 7.7-fold reduction of last level cache miss rate of Intel Nehalem, whereas register blocking increases data parallelism and thereby achieves 5.9 Gflops performance on a single core, which is over 55% of its peak performance. Register blocking+multithreading optimizations achieve 5.8-fold speedup on a single quadcore

Nehalem. Weak-scaling parallel efficiency is over 98% on 32,768 BlueGene/P processors.

3.1 High-Order Stencil Computation

This section first introduces HOSC, and then provides a detail of our application and experimental kernel used in the subsequent sections.

3.1.1 Stencil Computation

Accuracy of SC generally depends on the order of its stencil. Applications employing SC mostly involve partial differential equation solvers, which are based on finite-difference and multigrid methods, to study a vast array of phenomena including electromagnetic [89] and acoustic [76] wave propagation, and quantum dynamics [64]. Due to its pivotal role in computational sciences, SC is included in a number of benchmark suites such as PARKBENCH [75] and NAS Parallel Benchmarks [5]. Implementation of special purpose stencil compilers [8, 9, 85] and development of compiler optimizations [82] highlight the common use of SC based methods.

In SC, values $u^{(t)}(\mathbf{r})$ are assigned to a set of discrete grid points $\mathbf{\Omega} = \{\mathbf{r}\}$ for a number of simulation time steps $t \in [1, N_{step}]$. SC routine sweeps over $\mathbf{\Omega}$ iteratively to compute values of the grid points at next iteration, $u^{(t+1)}(\mathbf{r})$, as a function of the values of the neighboring nodal set at time t, $\mathbf{\Omega}' = \{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\}$, determined by the stencil geometry. According to the geometric arrangement of the nodal group *neighbor*($\mathbf{r}$), SC may be classified as follows: First, the order of a stencil, n, is defined as the distance between the grid point of interest, $\mathbf{r}$, and the farthest grid point in *neighbor*(**r**) along a certain axis. (In a finite-difference application, the order increases with required level of precision.) Second, we define the size of a stencil as the cardinality $|\{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\}|$, i.e., the number of grid points involved in each stencil iteration. Third, the footprint of a stencil is defined by the cardinality of minimum bounding orthorhombic volume, which includes all involved grid points per stencil. For example, Fig. 3.1 shows a 3rd order, 13-point SC, whose footprint is $7^2 = 49$ on a 2-dimensional lattice. Such stencil is widely used in high-order finite-difference calculations [90, 94]. In Fig. 3.1, the grid point of interest, **r**, is shown as a large solid circle while the set of neighbor points, excluding **r**, $\{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\} - \{\mathbf{r}\}$, is illustrated as small solid circles. Open circles show the other lattice sites within the stencil footprint, which are not used for calculation of $u^{(t+1)}(\mathbf{r})$.

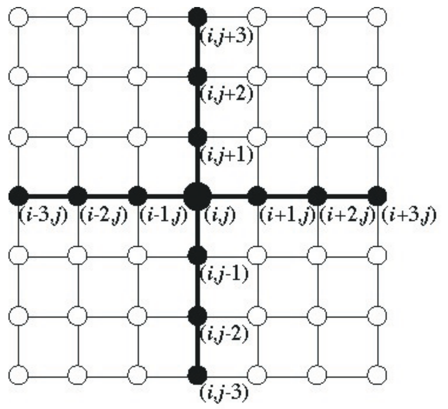


Figure 3.1: 3rd order, 13-point SC, whose footprint is 72 on a 2-dimensional lattice

A typical computation for updating the value of the central grid point shown in Fig. 3.1 is

$$\begin{aligned} u_{i,j}^{(t+1)} &= c_{-3} * u_{i-3,j}^{(t)} + c_{-2} * u_{i-2,j}^{(t)} + c_{-1} * u_{i-1,j}^{(t)} + c_{3} * u_{i+3,j}^{(t)} \\ &+ c_{2} * u_{i+2,j}^{(t)} + c_{1} * u_{i+1,j}^{(t)} + c_{-3} * u_{i,j-3}^{(t)} + c_{-2} * u_{i,j-2}^{(t)} \\ &+ c_{-1} * u_{i,j-1}^{(t)} + c_{3} * u_{i,j+3}^{(t)} + c_{2} * u_{i,j+2}^{(t)} + c_{1} * u_{i,j+1}^{(t)} + c_{0} * u_{i,j}^{(t)} \end{aligned} \tag{1}$$

where the subscripts i, j of $u^{(t)}$ represent the coordinates of grid points, and the coefficients are denoted as c. Time dependency is not shown on coefficients c, as they are constants in time but a function of the distance to the central grid point.

3.1.2 A High-Order Stencil Application for Seismic Wave Propagation

Our experimental SC application simulates seismic wave propagation by employing a 3D equivalent of the stencil in Fig. 3.1 to compute spatial derivatives on uniform grids using a finite-difference method. The 3D stencil kernel is highly off-diagonal (6th order) and involves 37 points (footprint is $13^3 = 2{,}197$), i.e., each grid point interacts with 12 other grid points in each of the x, y and z Cartesian directions.

A typical problem space may involve several hundreds of grid nodes in each dimension amounting to millions of grid points in total. The seismic application reported here uses $384^3 = 56.62$ million points. For each grid point, the code allocates 5 floats to hold temporary arrays and intermediate physical quantities and the result, therefore its memory footprint is 5×4 bytes $\times\ 384^3 = 1.13$ GBytes. Thus, the application not only involves heavy computations, but also needs a large amount of memory in comparison to the cache size offered by multicore architectures in the current HPC literature.

Table 3.1 shows the pseudocode of a computational kernel of the seismic application. In the kernel, the triply-nested loop in 3D Cartesian coordinates updates the value of each target grid points, $u^{(t+1)}$, based on values of neighboring grid points at time *t*, i.e., $u^{(t)}$. Stride in memory space is a critical factor to optimize SC application. In our kernel, the allocation for both $u^{(t)}$ and $u^{(t+1)}$ are 3 dimensional, 16-Bytes aligned, dynamic arrays, where x is unit stride direction, y is n_x stride, and z is the highest stride $n_x \times n_y$. The size of the problem domain is $(n_x - 2n) \times (n_y - 2n) \times (n_z - 2n)$, where n is the stencil order and n_x, n_y, and n_z are the numbers of grid points in the three Cartesian coordinates. The scopes of each loop are reduced by $2n$ (n = 6 in our case) to avoid complex boundary conditions from the experimental kernel.

Table 3.1: Pseudocode of the high-order stencil kernel.

```
procedure $naiveHOSC(n, n_x, n_y, n_z, c, u^{(t)}, u^{(t+1)})$
      for $i = n$ to $n_x - n$ do
         for $j = n$ to $n_y - n$ do
            for $k = n$ to $n_z - n$ do
               for $i_n = -n$ to $n$ do //X sweep
                  $u^{(t+1)}_{i,j,k} \leftarrow u^{(t+1)}_{i,j,k} + c_{i_n} * u^{(t)}_{i+i_n,j,k}$
               end for
               for $j_n = -n$ to $n$ do //Y sweep
                  $u^{(t+1)}_{i,j,k} \leftarrow u^{(t+1)}_{i,j,k} + c_{j_n} * u^{(t)}_{i,j+j_n,k}$
               end for
               for $k_n = -n$ to $n$ do //Z sweep
                  $u^{(t+1)}_{i,j,k} \leftarrow u^{(t+1)}_{i,j,k} + c_{k_n} * u^{(t)}_{i,j,k+k_n}$
               end for
            end for
         end for
      end for
end procedure
```

3.2 Hierarchical Parallelization

This section outlines our top-down approach to application tuning at system-, node- and microarchitecture-levels. We discuss our parallelization framework that combines: (1) inter-node parallelization via spatial decomposition; (2) intra-node parallelization via multithreading; (3) data locality optimizations through tiling; (4) RB and data parallelism via SIMD techniques to utilize registers; and (5) software prefetching to hide memory latency.

3.2.1 Inter-node Parallelism by Spatial Decomposition

Our parallel implementation essentially retains the computational methodology of the original sequential code. All of the existing subroutines and the data structures are retained. Our parallelization is based on spatial decomposition, where the 3D data space is decomposed into a mutually exclusive and collectively exhaustive set of subdomains. We distribute the data by assigning each subdomain to a compute node in the network, and employ an owner-computes rule.

In a parallel processing environment, it is vital to consider the time complexity of communication and computation dictated by the underlying algorithm, to achieve efficient parallelism. For a d-dimensional stencil problem of order n and global grid size $n_x \times n_y \times n_z$ to run on p processors, the communication complexity is $O(n((n_x \times n_y \times n_z)/p)^{(d-1)/d})$, whereas computation associated by the subdomain is proportional to the number of owned grid points, therefore its time complexity is $O(n(n_x \times n_y \times n_z)/p)$ (which is linear in n, assuming a stencil geometrically similar to that shown in Fig. 3.1). For a 3D problem, they reduce to $O(n((n_x \times n_y \times n_z)/p)^{2/3})$ and $O(n(n_x \times n_y \times n_z)/p)$, which are proportional to the surface area and volume of the underlying subdomain, respectively. Accordingly, subdomains are chosen to be the optimal orthorhombic box in the 3D domain minimizing the surface-to-volume ratio, $O((p/(n_x \times n_y \times n_z))^{1/3})$, for a given number of processors (e.g., a cubic subvolume, if the processor count is cube of an integer).

The pseudocode in Table 3.2 represents our spatial-decomposition approach. We implement the message passing using Message Passing Interface (MPI) [93]. A typical SC requires boundary grid points to be exchanged among processors. Therefore each subdomain is augmented with a surrounding layers of buffer grids used for data transfer.

Table 3.2: Pseudocode for spatial decomposition.

```
 1: for $t = 1$ to $N_{step}$
 2:     if $parity = sendfirst$
 3:         send sendBuffer to neighbor
 4:         receive recvBuffer from neighbor
 5:     else
 6:         receive recvBuffer from neighbor
 7:         send sendBuffer to neighbor
 8:     end if
 9:     forall $r \in \Omega_p$ do
10:         compute stencil in owned space
11:     end for
12: end for
```

In Table 3.2, the *parity* of a process determines if it first sends, i.e. a *sendfirst*, its own grid points at the boundary, *sendBuffer*, or receives the neighboring grid points from its neighbor process to update its receive buffer, *recvBuffer*. Once the buffer layers are exchanged, each process computes the stencil for their own grid points, i.e., $r \in \Omega_p$. Line 10 of Table 3.2 encapsulates the SC in lines 5-13 of Table 3.1.

3.2.2 Intra-node Parallelism by Multithreading

Our code is implemented based on hierarchical spatial decomposition: (1) inter-node parallelization with the higher-level spatial decomposition into domains based on message passing; and (2) intra-node parallelization with the lower-level spatial decomposition within each domain through multithreading. We implement 3 subroutines to handle x, y, and z directional sweeps inside the main loop shown in Table 3.1. The procedure *zSweepTh* in Table 3.3 shows a thread spawned to perform z directional sweep.

Table 3.3: Pseudocode for the threaded code section. Since threads store their data in the $u^{(t+1)}$ array, they avoid possible race conditions to exploit thread-level parallelism (TLP).

```
procedure $zSweepTh(n, n_{tx}, n_y, n_z, c, u^{(t)}, u^{(t+1)})$
       for $j = n$ to $n_y - n$ do
         for $i = n_{tx,l}$ to $n_{tx,u}$ do
           for $k = n$ to $n_z - n$ do
             $packed\_u_k^{(t)} \leftarrow u_{i,j,k}^{(t)}$
           end for
           for $k = n$ to $n_z - n$ do
             $u_{i,j,k}^{(t+1)} \leftarrow f(c, neighbor(packed\_u_k^{(t)}))$
           end for
         end for
       end for
end procedure
```

In Table 3.3, the *zSweepTh* thread receives n_{tx} as a parameter, which has lower and upper bounds of its assigned block, $n_{tx,l}$ and $n_{tx,u}$, respectively. This divides the global grid domain in the x direction. Recall that x is the unit-stride direction, y has n_x stride, and z has the highest stride of $n_x \times n_y$. Other threads reuse the method in *zSweepTh*, i.e., their innermost loop of the triply nested loop body is the direction of computation, second loop is the direction that has the lower stride among the other two Cartesian coordinates, and the outermost loop being the higher stride dimension. Notice that in line 5 of Table 3.3, we pack the grid points $u^{(t)}$ into $packed\_u^{(t)}$. This is in order to pay the non-contiguous memory access penalty only once for higher-stride dimensions, i.e., y and z. Therefore x thread does not implement this packing loop. In line 8, we represent the stencil to be a function f of coefficients c and the neighbor points of the grid point $packed\_u_k^{(t)}$. Here, the accessed neighbors are naturally the z neighbors since *zSweepTh* only implements the z-directional computation. Threads of the same direction can independently execute and perform updates on their assigned subdomain without introducing any locks until finally they are joined.

3.2.3 In-Core Optimizations

While it is possible to employ divide-and-conquer strategies for structured grid problems and perfectly scale such algorithms on massively parallel computers, utilizing in-core level optimization techniques is essential to exploit flops performance of the underlying computational units. In this section, we explore in-core optimization techniques addressing high-order stencil computations. Our tests include (1) loop unrolling and tiling targeting efficient L2 cache use; (2) register blocking (RB) and data-level parallelism via single-instruction multiple-data (SIMD) techniques to increase L1 cache efficiency; and (3) software prefetching to hide memory latency.

3.2.3.1 Loop Unrolling and Tiling

The stencil code in Table 3.1 has data reuse in all loops but traverses a very large memory footprint, which prevents the reuse to be fully exploited in the core's memory hierarchy. In this section, we discuss locality optimizations for exploiting data reuse.

To compute a single value, $u_{i,j,k}^{(t+1)}$, the stencil code reads 2*n* elements from each dimension of the 3-dimensional array $u^{(t)}$. The computation of $u_{i,j,k}^{(t+1)}$ in one of the dimensions reuses 2*n*-1 elements in that dimension and reads 2*n* new values in each of the other two dimensions. For example, the computation of $u_{i+1,j,k}^{(t+1)}$ reuses 2*n*–1 elements in x, and reads 2*n* new elements in y and z (plus a new element in x, $u_{i+1+n,j,k}^{(t)}$). Therefore each of the loops z, y and x reuses data across iterations. However, the data reused by different iterations of the outer loops may not be in cache(s) because of the large amount of data accessed in between.

The size of the memory footprint traversed by a stencil code is another factor preventing data locality. The memory footprint of $u^{(t)}$ in one iteration of loop x spans a memory region of $(2n-1)\times576$ KBytes, or 6.336 MBytes for $n = 6$, which is on the order of the shared data cache of most quadcore architectures. Assuming that each 3-dimensional array is allocated in a contiguous region of memory, two elements $u_{i,j,k}^{(t)}$ and $u_{i,j,k+1}^{(t)}$ are 576 KBytes apart. A few iterations of loop z touch more than a memory page, even for the large page size of the Nehalem.

To take advantage of the reuse available in the SC, we use code transformations that improve locality. Loop tiling is a code transformation that reorders loop iterations to bring accesses to the same data, cache line and memory pages closer together in time. We apply loop tiling to SC targeting different levels of memory hierarchy: TLB, last level cache (LLC), and SIMD register file. Before tiling, we apply loop unrolling to the three loops at the innermost level (i_n, j_n and k_n) of Table 3.1 with a factor of $2n$, i.e., we fully unroll the loops. Line 5 in Table 3.4 represents the computation in the loop body after unrolling. Fully unrolling these loops eliminates loop overhead and exposes more data reuse in the loop body that we will exploit using SIMD registers. The resulting code is a 3-deep loop nest, as shown in Table 3.4.

Table 3.4: Pseudocode for unrolled code. Function *f* represents stencil computation.

```
1: procedure $unrollHOSC(n, n_x, n_y, n_z, c, u^{(t)}, u^{(t+1)})$
2:         for $k = n$ to $n_z - n$ do
3:              for $j = n$ to $n_y - n$ do
4:                   for $i = n$ to $n_x - n$ do
5:                   $u_{i,j,k}^{(t+1)} \leftarrow f(c, neighbor(u_{i,j,k}^{(t)}))$
6:                   end for
7:              end for
8:         end for
9: end procedure
```

TLB: To reuse more data in each memory page and decrease the number of TLB misses, we apply tiling to loop *k*. Tile size *zTile* should be chosen such that the number of pages touched by loop iterations within the tile is smaller than the number of TLB entries.

LLC: The 8 MB L3 cache of our experimental platform (Intel Nehalem) can keep 13 “planes” of grid points, where a plane is a region of 384×384 elements (576 KBytes) along dimensions y and x (each value of z corresponds to a plane in (y, x)). However, to exploit reuse in all three dimensions, we tile loop *j* so that the data accessed within a tile of size *zTile* × *yTile* fits in the L3 cache (the data includes elements of $u^{(t)}$, $u^{(t+1)}$ and other temporary data required by the computation). Note that, although in this section we discuss locality optimizations in the context of a single thread on a single core, when combining these optimizations with parallelism, the tile sizes should be chosen carefully to prevent conflicts in the shared cache, i.e., L3 in 3-level cache Nehalem (L2 in earlier quadcore architectures).

Table 3.5 shows the SC after loop tiling is applied to loops *k* and *j*. In Table 3.5, macro *MIN* returns the smaller of its parameters, so that the two higher-stride directional loops, i.e., *y* and *z*, are subdivided into smaller loops of size *zTile* and *yTile*.

Table 3.5: Pseudocode for tiling implementation

```
 1: procedure $tiledHOSC(n,n_x,n_y,n_z,c,u^{(t)},u^{(t+1)})$
 2:        for $k=n$ to $n_z-n$ by $zTile$ do
 3:        for $j=n$ to $n_y-n$ by $yTile$ do
 4:           for $kk=k$ to $MIN((kk+zTile),(n_z-n))$ do
 5:           for $jj=j$ to $MIN((jj+yTile),(n_y-n))$ do
 6:                 for $ii=n$ to $n_x-n$ do
 7:                  $u^{(t+1)}_{ii,jj,kk} \leftarrow f(c,neighbor(u^{(t)}_{ii,jj,kk}))$
 8:                 end for
 9:             end for
10:            end for
11:        end for
12:        end for
13: end procedure
```

The tiling parameters, *zTile* and *yTile*, should be large enough to amortize the cost of the added loops but the size of the tiles should not exceed physically available cache size. In fact, for a 3-dimensional SC with m physical quantities (e.g., pressure and/or temperature) of type *datatype* per grid point, the tiling parameters should be chosen such that $yTile \times zTile \times m \times n \times sizeof(datatype) \leq$ effective cache size, where we refer to effective cache size instead of physical size to account for memory mapping rules.

In contrast to some existing tiling approaches that tile all three loops [53], the pseudocode in Table 3.5 does not apply tiling to the unit-stride dimension. Our tests have shown that tiling the two high-stride dimension loops is enough to have most reuse across tiles. In fact, tiling all three loops expands tile boundaries and cannot amortize the increased number of tile execution. This comes from the fact that in stencil codes, it is only required to preserve the group reuse at $2n+1$ distance.

Code generation and tuning: We use a code transformation tool, CHiLL [15], to generate optimized code variants of the SC. CHiLL is a transformation framework that allows the user to specify code transformations using a high-level script interface. CHiLL supports code transformations such as loop tiling, interchanging, unrolling, fusion and distribution, data copying and data prefetching. CHiLL takes as input the original code and a script specifying code transformations, and automatically generates a transformed code. Optimization parameters such as tile sizes can be specified as integer values or as unbound parameters to be determined later. In our optimized code variant (*stencil_2dTiling*) loops z and y are tiled with unbound parameters *zTile* and *yTile*. We use empirical search to determine the optimum tile sizes *zTile* and *yTile*, RB and SIMD parallelization.

3.2.3.2 Register Blocking

In SC, it is not possible to have small strides for all directions of computation at the same time. For example, the stride in the z direction is 576 KBytes in our kernel, whereas the x direction has a unit stride, i.e. 4 Bytes.

To exploit SIMD parallelism and spatial reuse in the x direction, we rearrange the computation to perform updates to grid points that are contiguous in the SIMDized code. Fig. 3.2 schematically represents our approach to RB for HOSC. In the original kernel in Fig. 3.2(a), the red square indicates target grid points to be updated at certain time. Blue, yellow and green squares respectively show memory locations of the nearest neighbor points in x, y, and z Cartesian coordinates. During each iteration of loop *ii* (in Table 3.5), the red square is updated by accessing all neighbors (not necessarily the nearest neighbors, and the stencil order determines the distance of the furthest accessed neighbor.) Memory stride for each direction is 1, n_x, and $n_x \times n_y$, respectively, using the same notation as in Table 3.5. In contrast to the original access pattern shown in Fig. 3.2(a), RB deals with a chunk of target grid points contiguous in the x direction. The same size of neighboring cells is fetched to update the block of target grids, which maximally utilize registers (see Fig. 3.2(b)).

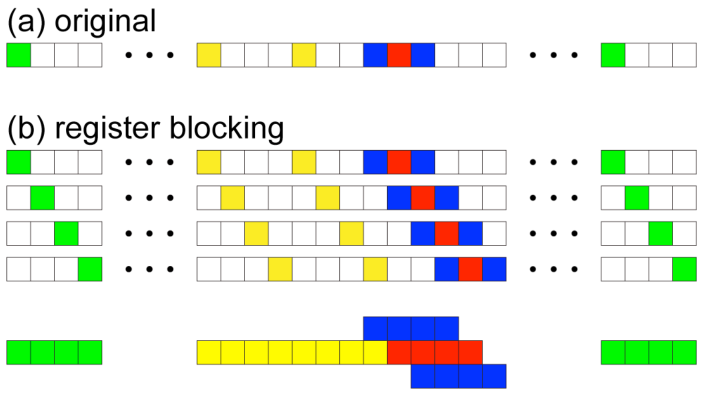


Figure 3.2: Memory access pattern of register blocking compared to the original access pattern

More specifically, with the RB technique, instead of computing $u_{i,j,k}^{(t+1)}$ one by one, we accumulate the contributions of $u_{i,j,k+n}^{(t)}$, $u_{i+1,j,k+n}^{(t)}$, $u_{i+2,j,k+n}^{(t)}$, $u_{i+3,j,k+n}^{(t)}$ on $u_{i,j,k}^{(t+1)}$, $u_{i+1,j,k}^{(t+1)}$, $u_{i+2,j,k}^{(t+1)}$, $u_{i+3,j,k}^{(t+1)}$, where both of 4 float blocks are contiguous and 16 Bytes aligned in memory, and can be packed into 16 Bytes SIMD registers. We manually implement RB using Intel SSE3 intrinsics. RB eliminates 3 memory accesses to neighboring grid points per 16-Byte block and accordingly increases performance. It should also be noted that the compiler fails to generate code using Intel SSE instructions for floating point operations in the tiling mode due to the complex loop body of high-order stencil. Therefore, RB enhances opportunity for better instruction scheduling with increased instruction level parallelism.

3.2.3.3 Prefetching

L2 data cache miss incurs the penalty of accessing to memory, which is approximately 150 clock cycles, an order of magnitude more than L2 access penalty, on modern quad-core architectures such as Intel Xeon processors. Therefore, we use software prefetching to hide memory latency.

We use Intel's `_mm_prefetch` intrinsic to implement fetching data from memory to second level cache. Since prefetch scheduling distance (PSD) is not a well-defined metric, and to achieve better performance, we spread the prefetch instructions inside the instruction sequence of the innermost loop rather than clustering prefetches together and experiment with a variety of PSDs up to 3 full iterations of the innermost loop. Considering high stride access due to the topology of stencil computation, we also combine this by translation lookaside buffer (TLB) priming to preload the page table entry for the z-neighboring grid points. This is similar to prefetch, but instead of a data cache line, the page table entry is being loaded in advance of its use to avoid TLB miss.

3.3 Performance Evaluation

Inter-node scalability tests have been carried out on (1) Intel Xeon and AMD Opteron platforms at the High Performance Computing and Communications (HPCC) Center of the University of Southern California (USC) and (2) the IBM BlueGene/P at Argonne National Laboratory. The Intel dual quadcore Xeon E5420 (Harpertown) processors are clocked at 2.5 GHz, featuring a 4×32 KBytes 8-way set associative L1 data and instruction cache and 2×6144 KBytes 24-way set associative L2 cache. The AMD dual dualcore Opteron 270 is clocked at 2 GHz with 1 MB L2 cache per core. The Xeon platform deploys 12 GB memory and 10-gigabit Myrinet interconnects, and the Opteron platform 4GB memory with 2-gigabit Myrinet. The BlueGene/P has four nodes on a chip, where each node has 2 GB DDR2 DRAM and four 450 POWER PC processors clocked at 850 MHz, featuring a 32 KBytes instruction and data cache, a 2 KBytes L2 cache and a shared 8 MB L3 cache. BlueGene/P also allows users to specify arrangement of MPI processes to make use of its 3D torus network topology. We have examined TXYZ and XYZT mapping orders, where XYZ represents 3D indices in the torus network while T (= 0, 1, 2, 3) corresponds the number of cores on one node. For a spatial decomposition scheme, though the best performance is expected by mapping process arrangement exactly on the torus structure [45], TXYZ mapping often performs better, retaining more communications locally.

We perform our single-processor and single-core benchmarks on Intel Nehalem (core i7 920), which is clocked at 2.67 GHz. On a single-die, quadcore Nehalem CPU, there is 8 MB of shared L3 cache, 256 KBytes of L2 cache per core and 64 KBytes of L1 cache (divided into a 32 KBytes instruction cache and a 32 KBytes data cache). Nehalem drops the front side bus (FSB) in favor of Intel QuickPath Interconnect (QPI) and, in doing so, brings the memory controller on-die. Cores on the Nehalem die communicate through QPI that offers 25.6 GB/s of bandwidth, which is more than twice the theoretical bandwidth of Harpertown's FSB. Nehalem's cores are capable of Simultaneous Multi-Threading (SMT), i.e., each core can execute 2 threads simultaneously, opposed to single thread per core on Harpertown.

3.3.1 Strong Scaling

Algorithms harnessing sound strong scalability may accelerate overall application performance by increasing the number of processors, and are desirable to fully utilize many-core architectures. Here, we define the strong-scaling speedup as the ratio of the time to solve a problem on one processor to that on *p* processors. Fig. 3.3(a) shows the total execution and communication times on BlueGene/P with TXYZ and XYZT network mappings as a function of the number of processors. Fig. 3.3(b) and Fig. 3.3(c) plot strong-scaling speedup on the three platforms described above. Measurements are done with a fixed global problem size of 400^3 grid points. We obtain good overall strong scalability by increasing processor count on all platforms. Furthermore, we observe superlinear speedups on the Intel architecture. This may be explained as an effect of aggregate cache size as discussed below.

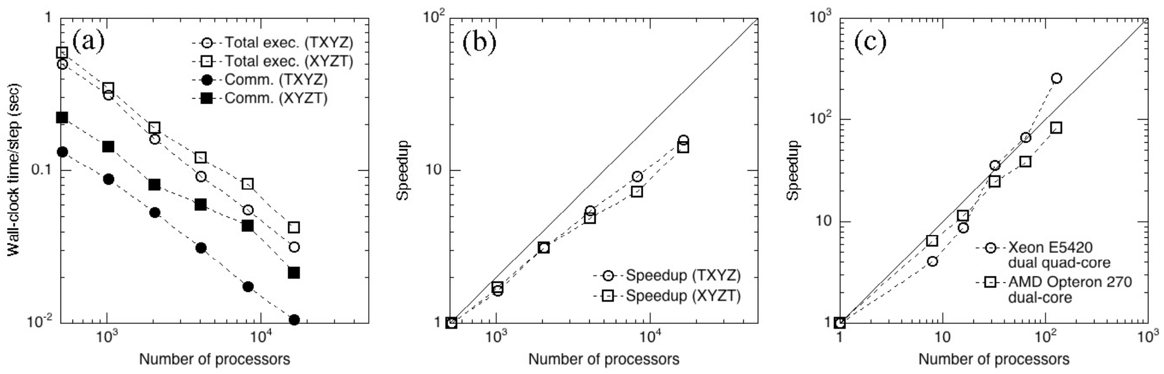


Figure 3.3: Strong-scaling benchmark on BlueGene/P, Intel Xeon E5420 and AMD Opteron 270 based clusters. (a) The wall-clock time as a function of the number of processors (up to 32,768) of BlueGene/P. (b) Speedup over 512 cores. (c) Speedup over a single core on the Xeon E5420 and Opteron 270 platforms. The Xeon E5420 exhibits superlinear speedup for relatively small number of processors.

Sequential or shared-memory implementations suffer from a large memory footprint per process. For example, our seismic application uses 384^3 grid points (1.13 GBytes data in total), which is well beyond the current cache sizes. The spatial decomposition scheme avoids this problem by increasing the processor count, and thus decreasing the volume of each subdomain. Substantial performance gain is expected, when the sub-domain size becomes small enough to fit into the cache. We use the Intel Vtune Performance Analyzer to monitor cache and TLB misses in the original seismic code. We find that high-stride computation accounts for more than 25% of core cycles throughout the thread execution during page-walks, indicating that a high TLB miss rate results in greater effective memory access time. The cache effect is most pronounced in the benchmark on Intel Xeon E5420 (Harpertown) architecture that features a shared 6 MBytes L2 cache per chip that accommodates two cores. This amounts to 12 MBytes of cache per multi-chip module (MCM), 6 times more than 2 MBytes (2×1 MBytes) L2 cache on AMD Opteron. Therefore, Harpertown outperforms AMD Opteron and shows the superlinear-scaling with the processor counts over 32 (see Fig. 3.3(c)).

3.3.2 Weak Scaling

Next, we test weak-scaling parallel efficiency of our parallel SC. We define the weak-scaling parallel efficiency as the running time on 1 processor divided by that on *p* processors. Fig. 3.4(a) and Fig. 3.4(b) plot the total execution time and communication time per SC step on the BlueGene/P and Intel Xeon E5345 (Clovertown) platforms.

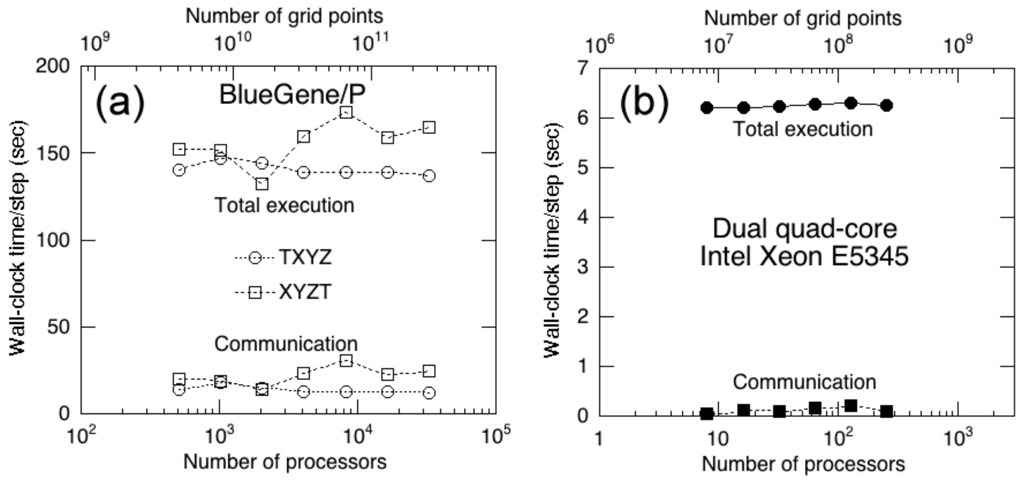


Figure 3.4: Weak-scaling performance on (a) BlueGene/P with 200^3 grid points per process and (b) a dual quadcore Intel Xeon E5345 cluster with 100^3 grid points per process

In the Fig. 3.4, the number of grid points per process (i.e. grain size) is 200^3 (160 MBytes/process) for BlueGene/P and 100^3 (20 MBytes/process) for the Clovertown cluster. We observe excellent weak scalability, nearly constant performance up to 256 processors on the Clovertown-based cluster and 32,768 processors on BlueGene/P with TXYZ network mapping. The XYZT network mapping shows fluctuations in the total and communication times, which may be due to higher possibility of interference with other processes.

3.3.3 Multithreading

In addition to the massive inter-node scalability demonstrated above, our parallelization strategy involves the lower levels of optimization: First, we use multithreading explained in section 3.2.2, implemented with POSIX thread. Next, we vectorize the loop for construction of $packed\_u_k^{(t)}$ and computation of the stencil

$f(c, neighbor(packed\_u_k^{(t)}))$ inside the triply nested for loops in Table 3.3 by using SSE3 intrinsics on the Intel Nehalem platform. We use Intel C compiler (icc) version 11 with O3 optimization level for our benchmarks.

Fig. 3.5(a) shows the reduction in clock time spent per simulation step due to multithreading and SIMD optimizations. Corresponding speedups are shown in Fig. 3.5(b).

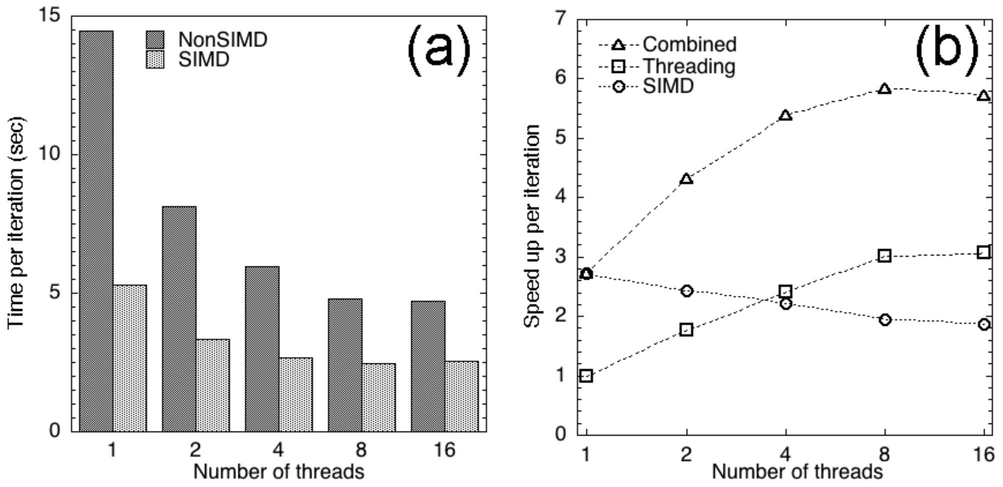


Figure 3.5: (a) The wall-clock time per iteration for non-SIMD and SIMDized codes as a function of the number of threads on quadcore Intel Nehalem. (b) Breakdown of the speedups due to multithreading and data-level parallelism along with the combined intra-node speedup. The best observed intra-node speedup is 5.83 with 8 threads on 4 cores.

So as to delineate the performances of multithreading and SIMDization, we define a performance metric as follows. We use the clock time for one simulation step of the single threaded, non-vectorized code to be the sequential run time T_s. We denote the parallel run time, $T_p(NUM\_THREADS)$, to be the clock time required for execution of one time step of the algorithm as a function of spawned thread number, in presence of both multithreading and SIMD optimizations. Then combined speedup, S_c, shown in Fig.

3.5(b) is the ratio T_s/T_p as a function of the number of threads. We remove SIMD optimizations to quantify the effect of multithreading only, and measure the parallel running times for a variety of thread numbers, and state the multithreading speedup, S_t, with respect to T_s. Finally, we attribute the excess speedup, S_c/S_t, to SIMD optimizations.

Fig. 3.5(b) shows the best speedup of 5.83 for 8 threads on a single quadcore Intel Nehalem node. It should be noted that multithreading speedup continues to increase until 8 threads on 4 cores. This is because Nehalem cores feature simultaneous multi-threading technology (SMT) that enables each core to run two threads at the same time. Increased L3 cache sharing and thread management cost dominates at 16 threads to yield moderate performance degradation.

3.3.4 Single-core Performance

We use empirical search to determine the best values of tile sizes *yTile* and *zTile* for *stencil_2dTiling* and its variant featuring RB, using a simple search on a 2-dimensional parameter space. The parameter space is defined by tile sizes <*yTile*, *zTile*>, bounded by cache and TLB capacity constraints. Our tiling approach essentially keeps cachelines closer to the core (in cache). All measurements are performed using one core of Intel Core i7 920 CPU. Table 3.6 summarizes the improved cache utilization by tiling.

Table 3.6: Performance comparison of original and tiled codes

Per 1000 instructions	Original	2dTiling	Improvement
LLC_MISS	0.42	0.054	7.77
Local DRAM	0.54	0.115	4.69
DTLB	0.41	0.084	4.88

As a reference, Table 3.6 lists the performance of the experimental kernel that incorporates loop-unrolling transformations labeled as original. Each row shows the number of CPU events per 1,000 retired instructions. The data in the rows are normalized with respect to 1,000 retired instructions during the code execution. The first row shows the number of retired loads that miss the LLC per 1,000 retired instructions at Nehalem core. We observe a 7.7-fold improvement in the LLC miss rate for the tiling implementation with the best tile sizes used. The second row shows number of memory load instructions retired where the memory reference missed the L1, L2 and LLC caches and required a local socket memory reference per 1,000 retired loads during the execution. We see that the tiling version of the code shows improvement by decreasing local memory access rate by a factor of 4.6. The third row shows the number of retired loads that missed the DTLB per 1,000 retired instructions. We also observe that tiling improves DTLB miss per 1,000 instructions by 4.8-fold.

Fig. 3.6(a) shows the variation of the performance (in terms of the floating-point operations per second) of *stencil_2dTiling* with tile sizes compared with that of the original loop-unrolling transformation code. We observe the dominant effect of y tile size, with the smaller y tile size achieving the better performance. The best observed performance is 2.72 Gflops, whereas the theoretical peak performance is 2.67×10^9 cycles/second $\times$ 4 flops/cycle = 10.68 Gflops. This corresponds to 25% of the peak performance. Auto-tuning through CHiLL generates similar flops performance.

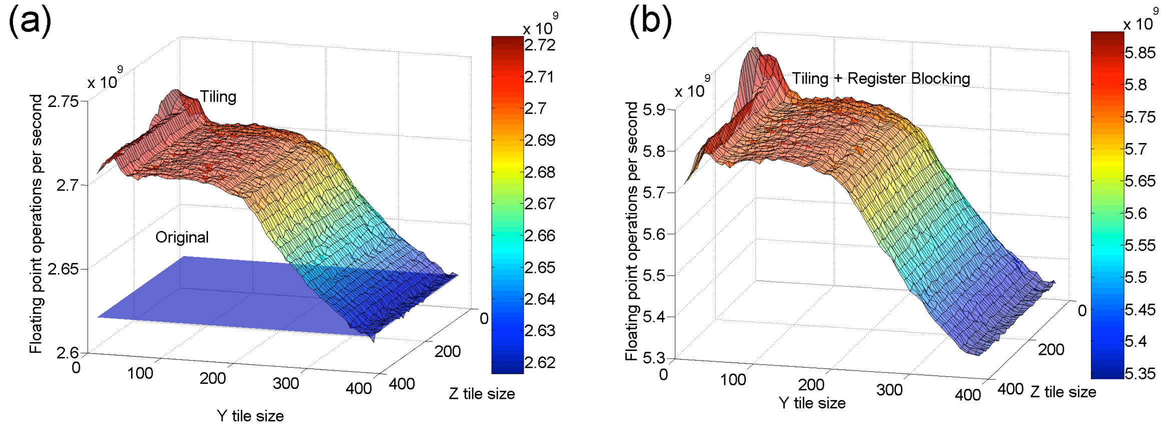


Figure 3.6: Floating-point operations per second performance at Intel Nehalem. (a) Comparison of the original and tiled codes. (b) The performance of tiling for TLB, cache and registers with SIMD implementation. The best single core performance is 5.9 GFlops.

Fig. 3.6(b) shows the tile-size dependency of the performance of the code variant incorporating both tiling and RB optimizations with the explicit use of SSE instructions. The best performance we have achieved is 5.9 Gflops, which corresponds to over 55% of the theoretical peak performance.

The above results show more than 2-fold improvement by the SSE featuring code variant with respect to the tiling version. This performance enhancement may be attributed to the effective use of Intel SSE instructions for floating-point operations. In fact, we use Intel C compiler (icc) version 11 for all codes with the best optimization level (-O3), but the compiler fails to vectorize the nested loop body for the HOSC. Our analysis with Vtune reveals that the SIMD variant reduces the number of instructions that retire the execution by more than 2-fold with respect to both original and tiling versions of the same code. To better understand this, we have examined the assembly codes for floating-point operations inside the loop body. Table 3.7 shows a typical assembly code for a single floating-point multiply generated from the original code and vector multiply of a four packed single-precision floating-point values in the SIMD code variant. Both codes load coefficient c and grid value $u_{i,j,k}^{(t)}$ and perform a multiply operation. Original code executes extra instructions to calculate array indices whereas SIMD variant uses 1 array index calculation per 4 floats multiply. Note the use of *unpcklps* and *movlhps* in the original code to unpack and move single-precision floating-point values, and use of *movups* for moving unaligned data. Also one single-precision floating-point multiply is performed by *mulss* instruction. In the SIMD code variant, *movaps* is used to load a 128-bit memory location to an XMM register at once, i.e., aligned load operation loads 4 float values at a time, then performs a packed multiply, *mulps*, to concurrently multiply 4 packed floats.

Table 3.7: Comparison of original and optimized code at the assembly level

Assembly for 1 float multiply from original code:

```
movss 0160(%rsp), %xmm14
mulss %xmm14, %xmm13
addss %xmm13, %xmm10
unpcklps %xmm8, %xmm8
movlhps %xmm8, %xmm8
movups 010(%rax, %r13, 4), %xmm11
mulps %xmm8, %xmm11
addps %xmm11, %xmm14
movss 0210(%rsp), %xmm6
mulss %xmm6, %xmm8
```

Assembly for 4 float vector multiply in SIMD code variant:

```
movaps (%rdx), %xmm10
movaps 010(%rbp, %rax, 4), %xmm0
mulps %xmm0, %xmm10
```

Lastly, we have explored performance enhancement thanks to software prefetching through micro-architecture level analysis of second-level cache utilization. Our results indicate that prefetching does not provide significant speedup. This is because the innermost loop in stencil kernel is predominately memory bandwidth-bound and already features competing techniques such as tiling and loop unrolling. It should also be noted that RB technique improves cache performance and this also contributes in less effectiveness of prefetching the data to L2 cache. Similar results were reported by Datta *et al.* for similar memory bound problems [18].

Chapter 4

A Metascalable Computing Framework

In the preceding chapter, we presented a scalable parallelization scheme for high-order stencil computations. In this chapter, we discuss a metascalable (or "design once, scale on new architectures") parallel computing framework that has been developed for large spatiotemporal-scale atomistic simulations of materials based on spatiotemporal data locality principles, which is expected to scale on emerging multipetaflops architectures. The framework consists of (1) an embedded divide-and-conquer (EDC) algorithmic framework based on spatial locality to design linear-scaling algorithms for high complexity problems; (2) a tunable hierarchical cellular decomposition (HCD) parallelization framework to map these O(*N*) algorithms onto a multicore cluster based on hybrid implementation combining message passing and critical section-free multithreading. The EDC-HCD framework exposes maximal concurrency and data locality, thereby achieving: (1) inter-node parallel efficiency well over 0.95 for 218 billion-atom molecular-dynamics and 1.68 trillion electronic-degrees-of-freedom quantum-mechanical simulations on 212,992 IBM BlueGene/L processors (superscalability); and (2) high intra-node, multithreading parallel efficiency (nanoscalability).

4.1 A Metascalable Dwarf

4.1.1 Embedded Divide-and-Conquer (EDC) Algorithmic Framework

In the embedded divide-and-conquer (EDC) algorithms, the physical system is divided into spatially localized computational cells [63]. These cells are embedded in a global field that is computed efficiently with tree-based algorithms (Fig. 4.1).

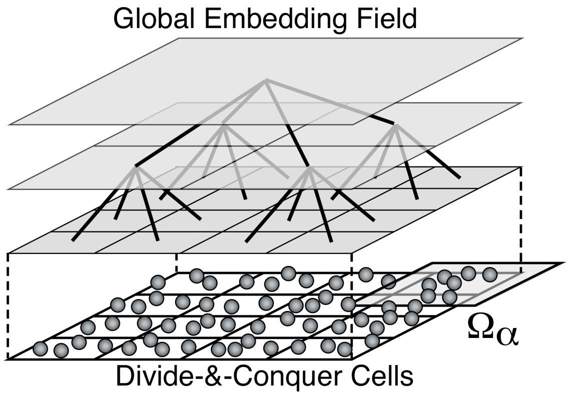


Figure 4.1: Schematic of embedded divide-and-conquer (EDC) algorithms. The physical space is subdivided into spatially localized cells, with local atoms constituting subproblems, which are embedded in a global field solved with tree-based algorithms.

Within the EDC framework, we have designed a number of O(N) algorithms (N is the number of atoms). For example, we have designed a space-time multiresolution MD (MRMD) algorithm to reduce the O(N^2) complexity of the N-body problem to O(N) [65]. MD simulation follows the trajectories of N point atoms by numerically integrating coupled ordinary differential equations. The hardest computation in MD simulation is the evaluation of the long-range electrostatic potential at N atomic positions. Since each evaluation involves contributions from $N-1$ sources, direct summation requires $O(N^2)$ operations. The MRMD algorithm uses the octree-based fast multipole method (FMM) [36, 73] to reduce the computational complexity to $O(N)$ based on spatial locality. We also use multiresolution in time, where temporal locality is utilized by computing forces from further atoms with less frequency [64].

We have also designed a fast ReaxFF (F-ReaxFF) algorithm to solve the $O(N^3)$ variable N-charge problem in chemically reactive MD in $O(N)$ time [69]. To describe chemical bond breakage and formation, the ReaxFF potential energy is a function of the positions of atomic pairs, triplets and quadruplets as well as the chemical bond orders of all constituent atomic pairs [100]. To describe charge transfer, ReaxFF uses a charge-equilibration scheme, in which atomic charges are determined at every MD step to minimize the electrostatic energy with the charge-neutrality constraint. This variable N-charge problem amounts to solving a dense linear system of equations, which requires $O(N^3)$ operations. The F-ReaxFF algorithm uses the FMM to perform the matrix-vector multiplications with $O(N)$ operations. It further utilizes the temporal locality of the solutions to reduce the amortized computational cost averaged over simulation steps to $O(N)$. To further speed up the solution, we use a multilevel preconditioned conjugate gradient (MPCG) method [60]. This method splits the Coulomb interaction matrix into far-field and near-field matrices and uses the sparse near-field matrix as a preconditioner.

The extensive use of the sparse preconditioner enhances the data locality, thereby increasing the parallel efficiency.

To solve the exponentially complex quantum N-body problem, we use an EDC density functional theory (EDC-DFT) algorithm mentioned in section 1.3.4.

4.1.2 Tunable Hierarchical Cellular Decomposition (HCD) for Algorithm-Hardware Mapping

To map the $O(N)$ EDC algorithms onto parallel computers, we have developed a tunable hierarchical cellular decomposition (HCD) framework.

Massively distributed scalability via message passing—Superscalability: Our parallelization in space is based on spatial decomposition, in which each spatial subsystem is assigned to a compute node in a parallel computer. For large granularity (the number of atoms per spatial subsystem, $N/D > 10^2$), simple spatial decomposition (i.e., each node is responsible for the computation of the forces on the atoms within its subsystem) suffices, whereas for finer granularity ($N/D \sim 1$), neutral-territory [87, 88] or other hybrid decomposition schemes [31, 79, 80, 92] can be incorporated into the framework. Our parallelization framework also includes load-balancing capability. For irregular data structures, the number of atoms assigned to each processor varies significantly, and this load imbalance degrades the parallel efficiency. Load balancing can be stated as an optimization problem [13, 20, 102]. We minimize the load-imbalance cost as well the size and the number of messages. Our topology-preserving spatial decomposition allows message passing to be performed in a structured way in only 6 steps, so that the number of messages is minimized. To minimize the load imbalance cost and the size of messages, we have developed a computational-space decomposition scheme [61]. The main idea is that the computational space shrinks in a region with high workload density, so that the workload is uniformly distributed. The sum of load-imbalance and communication costs is minimized as a functional of the computational space using simulated annealing. We have found that wavelets allow compact representation of curved partition boundaries and thus speed up the optimization procedure [59].

Multicore scalability via multithreading—Nanoscalablity: In addition to the massive inter-node scalability, "there is plenty of room at the bottom," as Richard Feynman noted. At the finest level, EDC algorithms consist of a large number of computational cells (Fig. 4.1), such as linked-list cells in MD [65] and domains in EDC-DFT [90], which are readily amenable to parallelization. On a multicore compute node, a block of cells is assigned to each thread for intra-node parallelization. Our EDC algorithms are thus implemented as hybrid message passing + multithreading programs. Here, we use the POSIX thread standard, which is supported across broad architectures and operating systems. In addition, our framework [63] includes the optimization of data and computation layouts [55, 96], in which the computational cells are traversed along various spacefilling curves [57] (e.g. Hilbert or Morton curve). To achieve high efficiency, special care must be taken also to make the multithreading free of critical sections. This is illustrated below using MRMD as an example.

In MRMD, the interatomic potential energy V consists of two-body (V_2) and three-body (V_3) terms [65]. To compute interatomic forces, the Newton's third law is usually used to reduce computation. However, for multithreading, a race condition occurs when computed force values are sent back to memory due to the random memory access pattern of MD application. We have designed a critical section-free algorithm to make all interatomic force computations independent. For example, Eq. (1) shows a conventional summation rule to compute the three-body interaction without duplicating the same calculation (the two-body computation follows a similar but simpler procedure):

$$V_3 = \sum_{i=1}^{N} \sum_{j<k}^{nbr(i)} v(\mathbf{r}_j, \mathbf{r}_i, \mathbf{r}_k), \tag{1}$$

where $\mathbf{r}_i$ is the coordinate of the i-th atom and $nbr(i)$ is the list of neighbor atoms within the three-body cutoff length from atom i. In order to avoid a race condition, we slightly rewrite Eq. (1) and calculate the three-body force, $\mathbf{F}_l^{(3)} = -\partial V_3 / \partial \mathbf{r}_i$, on the l-th atom as

$$\mathbf{F}_l^{(3)} = -\sum_{i=1}^{N} \sum_{j=1}^{nbr(i)} \sum_{k \neq i}^{nbr(j)} \frac{\partial v(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k)}{\partial \mathbf{r}_i} \delta_{i,l}, \tag{2}$$

where $\delta_{i,l} = 1$ ($i = l$) or 0 (else). Note that Eq. (1) has atom i as the center of atomic triplet (j, $\boldsymbol{i}$, k), whereas Eq. (2) has atom i as its head, ($\boldsymbol{i}$, j, k). The modified three-body interaction computation consists of a loop over atom i, each iteration of which traverses atoms j in the neighbor list of atom i and subsequently neighbor atoms k in $nbr(j)$. By assigning different head atoms i to different threads, there cannot be any race condition.

Our multithreading is based on a master/worker model, in which a master thread coordinates worker threads that actually perform force computations. We use POSIX semaphores to signal between the master and worker threads to avoid the overhead of thread creation and joining in each MD step. There are two check points at each MD time step, where all worker threads wait a signal from the master thread: (1) before the two-body force calculation loop, which also constructs the neighbor-lists, after atomic coordinates are updated; and (2) before three-body force calculation, after having all atoms complete neighbor-list construction. Table 4.1 and Fig. 4.2 show a pseudo-code and a flow chart of our algorithm, respectively.

Table 4.1: Pseudo-code of the critical section-free force computation algorithm

Master thread:

```
nstep = 0
compute 2-body and 3-body forces
spawn worker threads
while (nstep < Number_of_MD_steps)
  post semaphore₁ to worker threads to begin neighbor-list
construction and
    2-body force calculation
  wait for semaphore₁ signaled back from worker threads
  post semaphore₂ to worker threads to begin 3-body force
calculation
  wait for semaphore₂ signaled back from worker threads
  update atom positions
  nstep++
join all threads
```

Worker threads:

```
while (nstep < Number_of_MD_steps)
  wait for semaphore₁ from master thread
  start neighbor-list and 2-body force calculation
  post semaphore₁ to master thread
  wait for semaphore₂ from master thread
  start 3-body force calculation
  post semaphore₂ to master thread
  nstep++
```

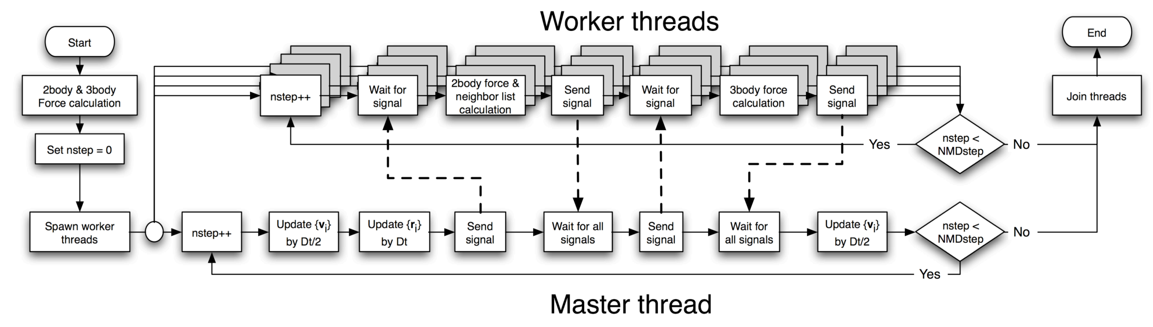


Figure 4.2: Flow chart of the critical section-free MD algorithm. At the beginning of simulation, a master thread computes complete force and energy before creating worker threads. Worker threads wait until master updates atomic coordinates $\{\mathbf{r}_i\}$ and velocities $\{\mathbf{v}_i\}$ (Dt is one MD time step). After receiving a signal from the master thread, the worker threads start computing two-body forces and neighbor-list, and then send a signal back to the master. Subsequently, three-body forces are computed following the same signaling scheme.

Intelligent tuning: The hierarchy of computational cells provides an efficient mechanism for performance optimization as well—we make both the layout and size of the cells as tunable parameters that are optimized on each computing platform [63]. Our EDC algorithms are implemented as hybrid message-passing + multithreading programs in the tunable HCD framework, in which the numbers of message passing interface (MPI) processes and POSIX threads are also tunable parameters. The HCD framework thus maximally exposes data locality and concurrency. We are currently collaborating with compiler and artificial intelligence (AI) research groups to use: (1) knowledge-representation techniques for expressing the exposed concurrency; and (2) machine-learning techniques for optimally mapping the expressed concurrency to hardware [6].

4.2 Scalability Tests

The scalability of our EDC-HCD applications has been tested on various high-end computing platforms including 212,992 IBM BlueGene/L processors at the Lawrence Livermore National Laboratory and 131,072 IBM BlueGene/P processors at the Argonne National Laboratory. The BlueGene/L comprises 106,496 compute nodes (CN), each with two IBM PowerPC 440 processors (at 700 MHz clock frequency) and 512 MB of shared memory. Each processor has a 32 KB instruction and data cache, a 2 KB L2 cache, and a 4 MB L3 cache, which is shared with the other processor on the CN. Each CN has two floating-point units that can perform fused multiply-add operations. In its default mode (co-processor mode), one of the processors in the CN manages the computation, while the other processor manages the communication. In an alternative mode of operation (virtual mode), both processors can be used for computation. A 3D torus network connects nearest-neighbor CNs, and a collective global tree network handles communications involving all nodes. The 163,840-processor BlueGene/P consists of 40 racks each with 32 node-cards. On every node-card, there are 32 quadcore nodes, each with 2 GB DDR2 DRAM and four 450 POWER PC processors sharing L3 cache. Compared to the BlueGene/L, the BlueGene/P has 20% higher clock frequency (850 MHz) and 1.4 times larger communication bandwidth (5.1 GB/s). While the sizes of the private L1 and L2 caches are the same on both platforms, the shared L3 cache of the BlueGene/P (8 MB) is twice as large as that of the BlueGene/L. On the BlueGene/P, hardware latency for nearest neighbors is less than 1 microsecond, and latency of one-way tree traversal is 1.3 microseconds.

4.2.1 Inter-node (Message-Passing) Spatial Scalability

Fig. 4.3 shows the execution and communication times of the MRMD, F-ReaxFF and EDC-DFT algorithms as a function of the number of processors *P* on the IBM BlueGene/L and P.

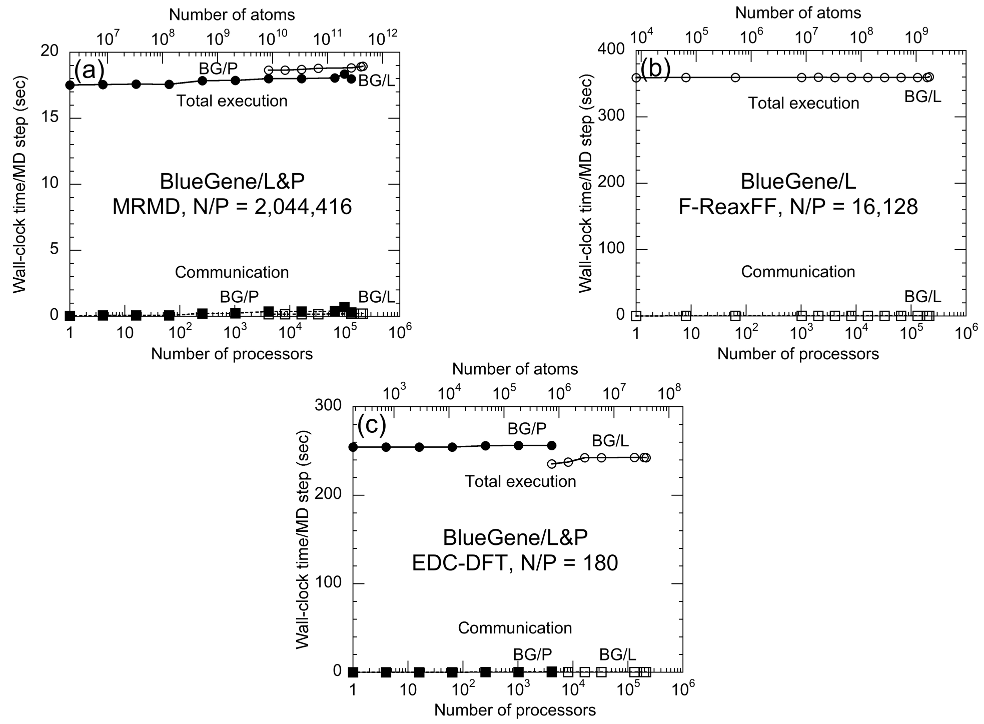


Figure 4.3: Total execution (circles) and communication (squares) times per MD time step as a function of the number of processors *P* of BlueGene/L (open symbols) and BlueGene/P (solid symbols) for three MD simulation algorithms: (a) MRMD for 2,044,416*P* atom silica systems; (b) F-ReaxFF MD for 16,128*P* atom RDX systems; and (c) EDC-DFT MD for 180*P* atom alumina systems.

In Fig. 4.3, we use one processor per dual-processor chip on BlueGene/L and all four cores per quadcore chip on BlueGene/P, so that 512 MB of memory is available per process to test the same problem size on both systems. Fig. 4.3(a) shows the execution time of the MRMD algorithm for silica material as a function of the number of processors

P. We scale the problem size linearly with the number of processors, so that the number of atoms N = 2,044,416P. In the MRMD algorithm, the interatomic potential energy is split into the long- and short-range contributions, and the long-range contribution is computed every 10 MD time steps. The execution time increases only slightly as a function of P on both BlueGene/L and P, and this signifies an excellent parallel efficiency. We define the speed of an MD program as a product of the total number of atoms and time steps executed per second. The isogranular speedup is the ratio between the speed of P processors and that of one processor. The weak-scaling parallel efficiency is the speedup divided by P, and it is 0.975 on 131,072 BlueGene/P processors. Though we have not completed all the benchmark runs, the measured weak-scaling parallel efficiency on 212,992 BlueGene/L processors is 0.985 based on the speedup over 4,096 processors. Fig. 4.3(a) also shows that the algorithm involves very small communication time.

Fig. 4.3(b) shows the execution time of the F-ReaxFF MD algorithm for RDX material as a function of P, where the number of atoms is N = 16,128P. The computation time includes 3 conjugate gradient (CG) iterations to solve the electronegativity equalization problem for determining atomic charges at each MD time step. On 212,992 BlueGene/L processors, the isogranular parallel efficiency of the F-ReaxFF algorithm is 0.996.

Fig. 4.3(c) shows the performance of the EDC-DFT based MD algorithm for 180P atom alumina systems. The execution time includes 3 self-consistent (SC) iterations to determine the electronic wave functions and the Kohn-Sham potential, with 3 CG iterations per SC cycle to refine each wave function iteratively. On 212,992 BlueGene/L processors, the isogranular parallel efficiency of the EDC-DFT algorithm is 0.998 (based on the speedup over 4,096 processors).

Our largest benchmark tests include 217,722,126,336-atom MRMD, 1,717,567,488-atom F-ReaxFF, and 19,169,280-atom (1,683,216,138,240 electronic degrees-of-freedom) EDC-DFT calculations on 212,992 BlueGene/L processors. Though the absolute strong scaling (i.e. solving the same problem from P = 1 to 212,992) is not meaningful for such massive parallelism, the three algorithms exhibit excellent differential strong scalability, i.e., insensitivity of the speed on the granularity N/P. To quantify this effect, Fig. 4.4(a) plots the program speed (measured by the product of the number of atoms and MD time steps executed per second) for the MRMD algorithms as a function of the granularity (the number of atoms per processor, N/P) on BlueGene/L. The program maintains the same speed over a wide range of granularity for each P. The same data points are shown in Fig. 4.4(b) as a function of the number of processors, P. The MRMD program thus achieves a nearly perfect linear speedup as a function of the number of processors independent of the value of N/P.

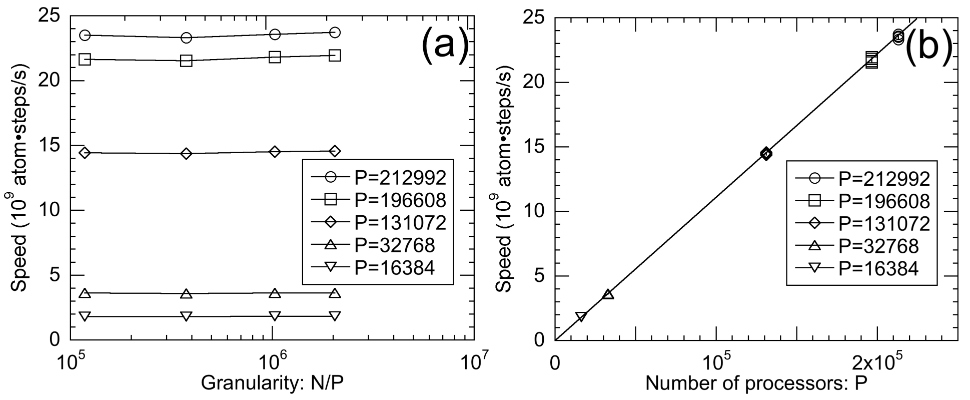


Figure 4.4: (a) Speed of MRMD on BlueGene/L as a function of the granularity *N*/*P*, for different numbers of processors *P*. (b) Speed of MRMD on BlueGene/L as a function of *P* for various granularities ranging from *N*/*P* = 117,912 to 2,044,416.

4.2.2 Intra-node (Multithreading) Spatial Scalability

We have tested the multithreading scalability of MRMD on a dual Intel Xeon quadcore platform. Fig. 4.5 shows the speedup of the multithreaded code over the single-thread counterpart as a function of the number of worker threads. In addition to the speedup of the total program, Fig. 4.5 also shows the speedups of the code segments for two-body and three-body force calculations separately. We see that the code scales quite well up to 8 threads on the 8-core platform. We define the multithreading efficiency as the speedup divided by the number of threads. The efficiency of two-body force calculation is 0.927, while that for three-body force calculation is 0.436, for 8 threads. The low efficiency of the three-body force calculation may be due to the redundant computations introduced in Eq. (2) to eliminate critical sections. Nevertheless, the efficiency of the total program is rather high (0.811), since the fraction of the three-body calculation is about one third of the two-body force calculation. This result shows that the semaphore-based signaling between master and worker threads is highly effective. In a test calculation for a 12,228-atom silica system, the running time is 13.6 milliseconds per MD time step.

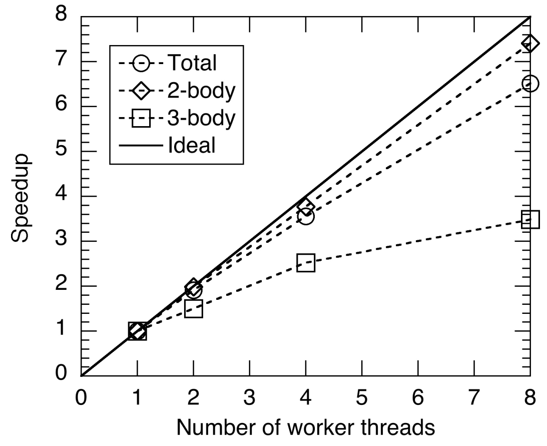


Figure 4.5: Speedup of the multithreaded MRMD algorithm over a single-threaded counterpart for the total program (circles), the two-body force calculation (diamonds), and three-body force calculation (squares). The solid line shows the ideal speedup.

We have thus developed high-end reactive atomistic simulation programs to encompass large spatiotemporal scales with common algorithmic and computational frameworks based on spatiotemporal data-locality principles. In fact, the metascalable dwarf can reduce diverse applications (including all of the original seven dwarfs) to a highly scalable form by common techniques of embedding and divide-and-conquer. According to the scalability tests presented in this chapter, they are likely to scale on future architectures beyond petaflops. The simulation algorithms are already enabling million-to-billion atom simulations of mechano-chemical processes, which have applications in broad areas such as energy and environment [17, 70, 97].

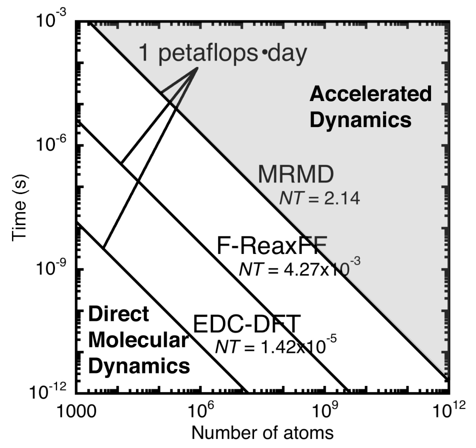


Figure 4.6: Spatiotemporal scales *NT* accessible by direct molecular-dynamics (white background) and approximate accelerated-dynamics (gray) simulations with a petaflops•day of computing. The lines are the *NT* achieved per petaflops•day of computing for MD (MRMD), chemically reactive MD (F-ReaxFF), and quantum-mechanical MD (EDC-DFT) simulations, respectively.

A critical issue, however, is the time scale studied by MD simulations. We define the spatiotemporal scale, *NT*, of an MD simulation as the product of the number of atoms *N* and the simulated time span *T*. On petaflops computers, direct MD simulations can be performed for *NT* = 1-10 atom•seconds (i.e. multibillion-atom simulation for several nanoseconds or multimillion-atom simulation for several microseconds). More specifically, a day of computing on a sustained petaflops computer (i.e. one petaflops•day of computing) achieves *NT* = 2.14 (e.g. 1 million atoms for 2.14 microseconds) (Fig. 4.6), according to the benchmark test in section 4.2 (i.e., extrapolated from the measured MRMD performance on the BlueGene/L, which is rated as 0.478 petaflops according to the Linpack benchmark). Accelerated-dynamics simulations [101] such as molecular-kinetics simulations [58] will push the spatiotemporal envelope beyond *NT* = 10, but they need to be fully validated against direct MD simulations at NT = 1-10. Such large spatiotemporal-scale atomistic simulations are expected to advance scientific knowledge.

Chapter 5

Optimizing *O*(*N*) Density Functional Theory

In preceding chapters, we have presented our parallelization framework for atomistic simulations and detailed our hierarchical optimization scheme for stencil computations. In this chapter, we unify our preceding discussions in performance analysis, multilevel parallelization and optimization to improve the computational performance of a production level first-principles molecular-dynamics application [90, 91]. This scientific code implements a divide-and-conquer algorithm based MD simulation that uses small DFT calculations “on the fly” for computing interatomic forces [48]. Here, we particularly focus on improving the scalability and floating point performance of this application on large-scale clusters, in particular on BlueGene/P and Intel Xeon clusters.

5.1 Application Description

The density functional theory (DFT) problem is formulated as a minimization of the energy functional $E_{\mathrm{QM}}[\psi^{N\mathrm{el}}]$ with respect to electronic wave functions (or Kohn-Sham orbitals) $\psi^{N\mathrm{el}}(\mathbf{r}) = \{\psi_n(\mathbf{r})|n = 1,...,N_{\mathrm{el}}\}$, subject to orthonormality constraints (N_{el} is the number of wave functions on the order of N, and $\mathbf{r}$ denotes a three-dimensional position vector) [43]. The asymptotic computational complexity of the DFT problem is $O(N^3)$ for the orthogonalization of the Kohn-Sham orbitals, while it is proportional to N^2 for iterative minimization methods for system sizes studied practically ($N < 10^3$, where the orthogonalization computation is not dominant) [40]. The data locality principle called quantum nearsightedness [49] in DFT is best implemented with a divide-and-conquer (DC) algorithm [105, 106], which naturally leads to $O(N)$ DFT calculations [35]. However, it is only in the past several years that $O(N)$ DFT algorithms, especially with large basis sets ($> 10^4$ unknowns per electron, necessary for the transferability of accuracy), have attained controlled error bounds, robust convergence properties, and energy conservation in its use in molecular dynamics (MD) simulations, to make large DFT-based MD simulations practical [29, 91]. The embedded divide-and-conquer (EDC) DFT algorithm considered in this research combines a hierarchical grid technique with multigrid preconditioning and adaptive fine mesh generation [90, 91]. Our EDC-DFT method is implemented as a ~38,500 lines Fortran 90 program that uses MPI for inter process message shuttling.

5.2 Systematic Optimization

In this section, we detail our systematic optimization approach. We first present our system- and node-level performance evaluation of the EDC-DFT code, and then detail our performance enhancement methodology.

5.2.1 Performance Evaluation

The performance analysis of our Density Functional Theory (DFT) based Molecular Dynamics (MD) code on BlueGene/P has revealed that our implementation has excellent weak- and strong-scaling properties.

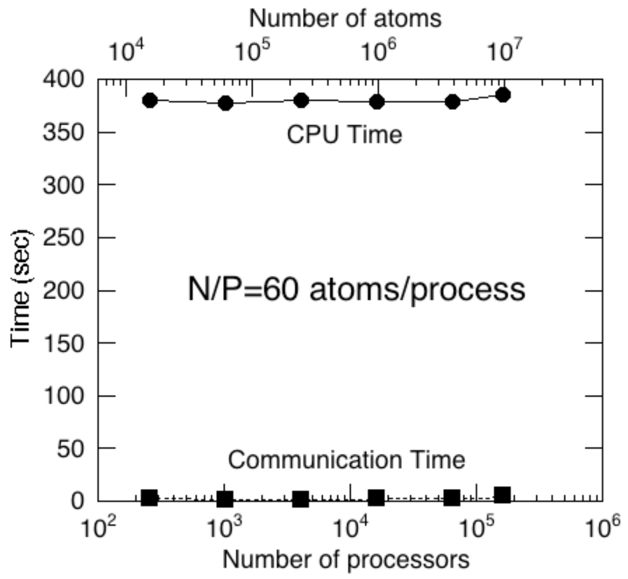


Figure 5.1: Weak-scaling performance

Fig. 5.1 shows the weak-scaling performance of the DFT code. Here, the principal x-axis shows the number of processors (*P*) used. The largest test case includes the whole BlueGene/P system with 163840 processors at Argonne National Laboratory. The secondary x-axis shows the total number of the atoms in the system (*N*). Here, we scale *N* linearly with *P*, where the grain size (*N*/*P*) is 60 atoms/process. We show both the computation (or CPU) time and the communication time in Fig. 5.1, where the time is measured by `MPI_WTIME()` thereby refers to the wall-clock time. The weak-scaling parallel efficiency, which we define as the run time on the smallest benchmark divided by that on increasing number of processors, is observed to be more than 98% on P = 163840. Thus our spatial decomposition implementation is highly effective in terms of inter-node scalability of DFT computations up to 163840 processors.

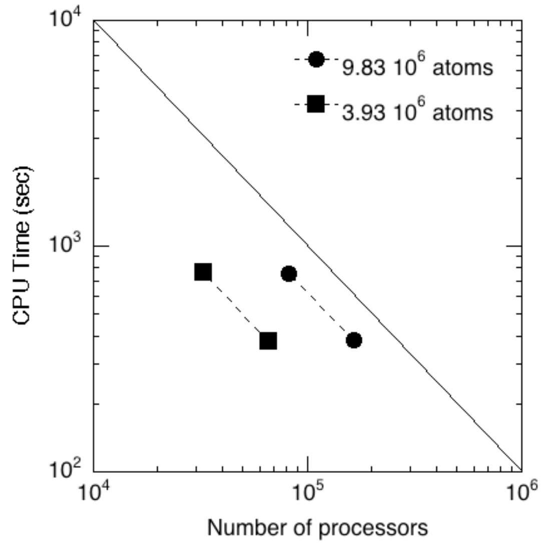


Figure 5.2: Strong-scaling performance

Fig. 5.2 shows the strong-scaling performance of the DFT code. In this benchmark, we keep *N* constant, and use different *P* to simulate the same physical system. Figure 5.2 plots the CPU times as measured by `MPI_WTIME()` as a function of *P*. The solid diagonal line shows the ideal strong-scaling behavior. We demonstrate two test systems composed of 9.83 and 3.93 million atoms. The largest test case uses the whole BlueGene/P system with 163840 processors. The strong-scaling parallel efficiency, which we define as the normalized ratio of the run times to solve the problem using larger number of processors, is observed to be more than 97%. This result, together with the weak-scaling performance, confirms that our spatial decomposition based divide-and-conquer implementation for *O*(*N*) scaling DFT problem almost ideally scales on BlueGene/P system and will continue to be scalable on the next generation BlueGene/Q computer.

We used the Tuning and Analysis Utilities (TAU) toolkit to gather performance information through the execution of our program on BlueGene/P.

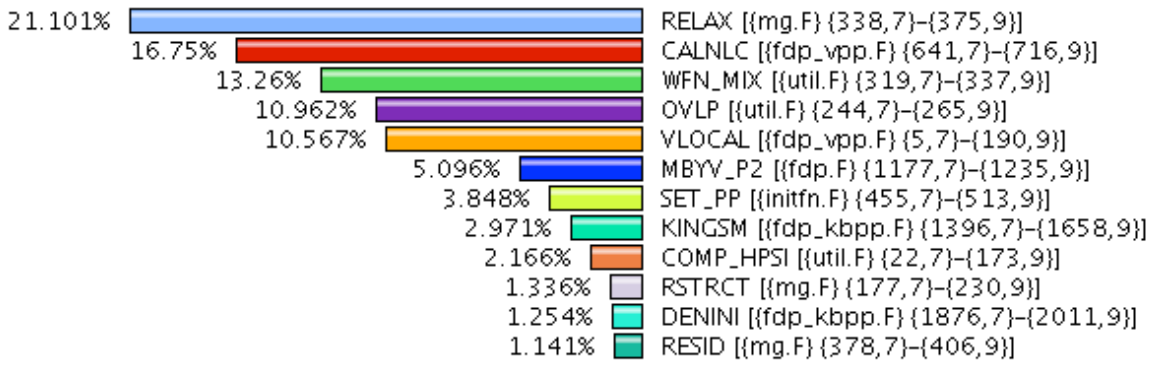


Figure 5.3: Percentage exclusive times

In Fig. 5.3, we show percentages of the exclusive times spent in routines averaged over a 64 processor run (in virtual-node mode, i.e., 4 MPI processes per node) with 60 atoms/processor grain size. In this figure, we demonstrate routines that have higher than 1% impact on the total execution time.

In Fig. 5.4, we present percentage exclusive times across each MPI process for this 64 processor run. Here, we have observed that the normalized standard deviation of the individual times for the shown routines on different processes is less than 1%. This shows that our spatial decomposition implementation divides the computational load evenly among the processors.

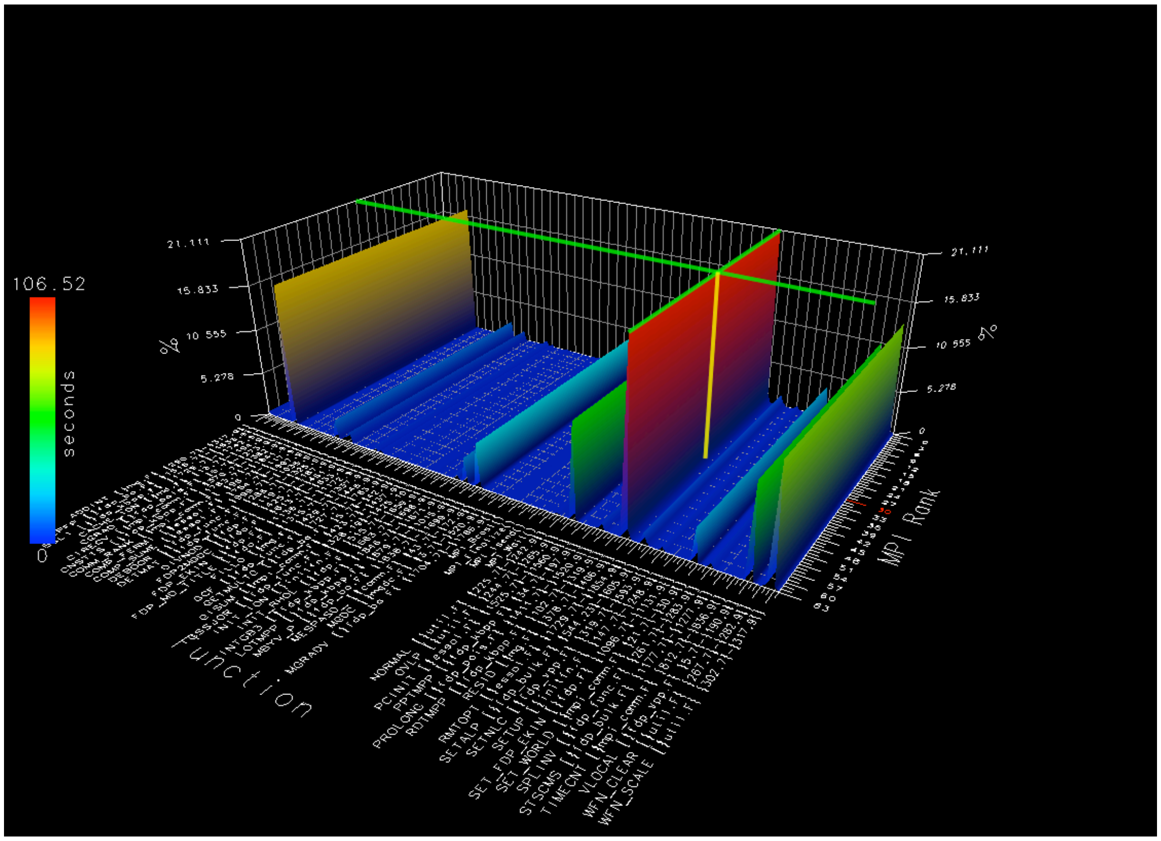


Figure 5.4: Percentage exclusive times across a 64-process run on BlueGene/P

In order to evaluate intra-node performance, we have identified the top performance bottleneck routines. Specifically, we have observed that the 4 most time-consuming subroutines given below spend more than 60% of the total CPU time. Therefore, in-core optimization of the corresponding subroutines could significantly reduce computation time.

1. Conjugate gradient relaxation

2. Sparse-matrix to vector multiplication

3. Linear mixing of wave functions with coefficients

4. Inner product of two wave functions

These subroutines mainly implement 2 computational kernels: (1) Nearest-neighbor and high-order stencil computations; and (2) Basic linear algebra operations such as vector dot product, and constant times a vector plus a vector.

In order to analyze the current floating point performance of these code sections, we have used IBM Hardware Performance Monitoring (HPM) API on BlueGene/P system. Our analysis with HPM revealed that the IBM XL Fortran compiler fails to generate instructions to utilize the dual floating-point units (FPUs) of the PowerPC (PPC) 450 processor. Dual FPU instructions operate on pairs of double-precision (DP) floating-point numbers, and perform single-instruction multiple data (SIMD) processing to deliver two DP floating-point operations per cycle. If instructions are not SIMDized, attainable performance is halved. In fact, our HPM analysis has shown that the floating-point operations per second (flops) performance of our code is ~200 Mflops per core, i.e., ~6% of the peak per-core flops performance of the PPC 450 processor. Finally, while we present our performance evaluation on BlueGene/P using TAU and HPM in this section, we have verified the validity of these results using Intel Vtune Performance Analyzer on a quadcore Intel Xeon 5320 processor as well.

5.2.2 Model-Guided Optimization

As mentioned in section 1.3.4, the EDC-DFT algorithm represents the physical system as a union of overlapping spatial domains, $\Omega = \cup_\alpha \Omega_\alpha$ (see Figure 5.5), and physical properties are computed as linear combinations of domain properties [90, 91].

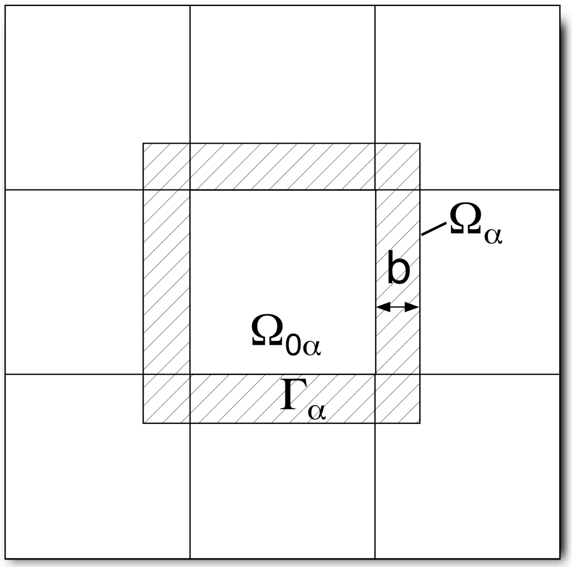


Figure 5.5: Schematic of the divide-and-conquer algorithm in 2D. The physical space Ω is a union of overlapping domains, $\Omega = \cup_\alpha \Omega_\alpha$. Each domain Ω_α is further decomposed into the non-overlapping core $\Omega_{0\alpha}$, and the buffer layer Γ_α (see the shaded area). The depth of the buffer layer is *b*.

For example, the electronic density is expressed as $\rho(\mathbf{r}) = \Sigma_\alpha p^\alpha(\mathbf{r}) \Sigma_n f_n^\alpha |\psi_n^\alpha(\mathbf{r})|^2$, where $p^\alpha(\mathbf{r})$ is a support function that vanishes outside the α-th domain Ω_α, and f_n^α and $\psi_n^\alpha(\mathbf{r})$ are the occupation number and the wave function of the *n*-th Kohn-Sham orbital in Ω_α. The domains are embedded in a global Kohn-Sham potential, which is a functional of $\rho(\mathbf{r})$ and is determined self-consistently with $\{f_n^\alpha, \psi_n^\alpha(\mathbf{r})\}$. We use the multigrid method to compute the global potential in $O(N)$ time. The DFT calculation in each domain is performed using a real-space approach [14, 95], in which electronic wave functions are represented on grid points. The real-space grid is augmented with coarser multigrids to accelerate the iterative solution. Furthermore, a finer grid is adaptively generated near every atom, in order to accurately operate ionic pseudopotentials for calculating electron-ion interactions.

Each domain Ω_α is further decomposed into its sub-volumes,

$$\Omega_\alpha = \Omega_{0\alpha} \cup \Gamma_\alpha, \tag{1}$$

where $\Omega_{0\alpha}$ is the non-overlapping core,

$$\Omega = \bigcup_\alpha \Omega_{0\alpha}; \quad \Omega_{0\alpha} \cap \Omega_{0\beta} = 0 \quad (\alpha \neq \beta) \quad , \tag{2}$$

and Γ_α is the buffer layer [21]. Boundary conditions on $\{\psi_n^\alpha(\vec{r})\}$ are imposed at the boundary $\partial\Omega_\alpha$ of domain Ω_α. We use either the rigid-wall boundary condition, in which the wave function vanishes at the boundary, or the periodic boundary condition. The wave function values in the buffer layer Γ_α may be contaminated by the artificial boundary conditions imposed at $\partial\Omega_\alpha$. Thus, the domain support function $p^\alpha(\vec{r})$ is made zero near $\partial\Omega_\alpha$ within Γ_α [21], so that the contaminated wave function values do not contribute to the density $\rho(\vec{r})$.

The EDC-DFT algorithm on the hierarchical real-space grids is implemented on parallel computers based on spatial decomposition [90, 91]. Each compute node contains one or more domains of the EDC algorithm. Then, only the global density but not individual wave functions needs to be communicated. The resulting large computation/communication ratio makes this approach highly scalable. In the following, we present our model-guided optimizations for the parallel EDC-DFT algorithm.

5.2.2.1 Optimization of Domain Size

We first optimize the spatial decomposition domain size in the EDC-DFT algorithm, based on an analysis of its computational cost given in Theorem 1.

Theorem 1: Consider a cubic system of side length L, and let the side length and the buffer depth of a domain in the DC-DFT algorithm be l and b, respectively. The computational complexity of the DFT computation within each domain is denoted as ν. Then, the optimal domain size l_*, which incurs the minimal computational cost, is given by

$$l_* = \frac{2b}{\nu - 1}. \tag{3}$$

Proof: The number of domains is $N_{\text{domain}} = (L/l)^3$, and the computational cost per domain can be written as

$$T_{\text{domain}} = c\left(l + 2b\right)^{3\nu}, \tag{4}$$

where c is a prefactor. The total computational cost is thus a function of l:

$$T_{\text{comp}}\left(l\right) = T_{\text{domain}} \bullet N_{\text{domain}} = c\left(\frac{L}{l}\right)^3 \left(l + 2b\right)^{3\nu}. \tag{5}$$

The optimal domain size to minimize the computational cost is then given by

$$l_* = \arg\min\left[T_{\text{comp}}\left(l\right)\right] = \frac{2b}{\nu - 1}. \tag{6}$$

QED.

As mentioned earlier, the computational complexity of the DFT problem is N^2 for typical domain sizes, $N < 100$, and thus $l_* = 2b$. (The asymptotic complexity, which has rarely been encountered in practical DFT calculations [39], is $O(N^3)$, and accordingly, $l_* = b$.) Based on the analysis presented here, we select optimal l_* in the b to $2b$ range in our typical run.

5.2.2.2 Optimization of Buffer Depth

The choice of the buffer depth b is dictated by accuracy requirement. Namely, the quantum nearsightedness principle [49] indicates that the error involved in the EDC-DFT algorithm decays exponentially as a function of b [81]. This proposition is confirmed by numerical experiments shown in Fig. 5.6 [90].

To test the convergence of calculation with respect to the buffer depth, Fig. 5.6 shows the calculated potential energy as a function of b for an amorphous CdSe system containing 512 atoms in a cubic cell of length 45.664 a.u. (The domain size is fixed as 11.416 a.u.) The potential energy converges within 10^{-3} a.u. per atom above $b = 4$ a.u.

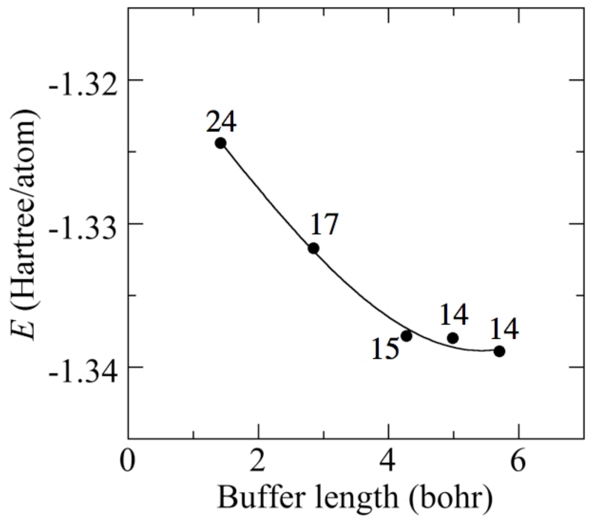


Figure 5.6: Potential energy as a function of the buffer length b for an amorphous CdSe system (512 atoms in a cubic cell of side length, 45.664 a.u.). The domain size is fixed as 11.416 a.u. The atomic units are used for both energy and length. Numerals in the figure indicate the number of self-consistent iterations required for the convergence of the electron density within $\left\langle \left\{ \left(\rho_i(\vec{r}) - \rho_{i-1}(\vec{r}) \right) / \rho_0 \right\}^2 \right\rangle \leq 10^{-4}$, where $\rho_i(\vec{r})$ is the electron density at ith iteration, ρ_0 is the average electron density, 0.0215 a.u., and the brackets denote the average over the grid for the entire system.

Since the computational complexity of the EDC-DFT algorithm scales with the buffer depth b asymptotically as $b^{3_\nu} = b^6 \sim b^9$ (see equation 5), the large b value required for obtaining a sufficient accuracy in energy (*e.g.*, 10^{-3} a.u. per atom) represents a major computational bottleneck. Based on the analysis presented here, we select the buffer depth parameter with respect to our accuracy requirements for different physical simulations.

5.2.3 Profiling-Guided Optimization

In this section, guided by our performance evaluation given earlier in section 5.2, we develop an efficient computation acceleration subsystem to reduce performance footprint of kernels mentioned earlier. The numerical core of DFT application represents a HOSC as discussed in section 3.1.1 therefore we utilize some of the in-core optimization methods that are detailed in section 3.2.3. More specifically, our approach includes the following methods to efficiently harvest computer processor and memory resources.

5.2.3.1 Performance Libraries

IBM Engineering and Scientific Subroutine Library (ESSL) and Intel Math Kernel Library (MKL) provide a collection of high performance mathematical subroutines for many common scientific and engineering applications. In our work, we took advantage of such performance libraries whenever possible. Specifically, `WFN_MIX` and `OVLP` subroutines shown in Fig. 5.3 implement linear mixing of wave functions with coefficients and inner product of two wave functions (represented as a linear array of float numbers), respectively. Both MKL and ESSL features subroutines that implement such functionality. In particular, we used `DAXPY` and `DDOT` subroutines provided by Intel MKL as an alternative to our existing `WFN_MIX` and `OVLP` subroutines, respectively.

5.2.3.2 Loop Blocking and Scalar Replacement

Our percentage exclusive times plot, given in Fig. 5.3, revealed that the conjugate gradient relaxation subroutine took %21.1 of the execution time averaged over our experimental runs. Based on this evaluation, we developed code transformations that improve data locality within the main loop inside this subroutine, as shown in Table 5.1.

Table 5.1: A sample code snippet from conjugate gradient relaxation subroutine optimization is shown. The first section represents the original implementation; bottom code snippet shows our scalar replacement and loop blocking implementation.

Original implementation:

```
ndat = nx1*ny1*nz1
      …
do n = 1, ndat
         i = ncorsx(n)
         j = ncorsy(n)
         k = ncorsz(n)
         km1 = k - 1
         kp1 = k + 1
         jm1 = j - 1
         jp1 = j + 1
         im1 = i - 1
         ip1 = i + 1
         g(n) = g(n) &
& - dk(1,dpth)*(wrk(im1,j,k)+wrk(ip1,j,k)) &
& - dk(2,dpth)*(wrk(i,jm1,k)+wrk(i,jp1,k)) &
& - dk(3,dpth)*(wrk(i,j,km1)+wrk(i,j,kp1))
end do
```

Scalar replacement and loop blocking implementation:

```
n = 1
     tmp1 = dk(1,dpth)
     tmp2 = dk(2,dpth)
     tmp3 = dk(3,dpth)
do k = 1, nz1
  do j = 1, ny1
     af = wrk(0,j,k)
     bf = wrk(1,j,k)
     cf = wrk(2,j,k)
    do i = 1, nx1
        km1 = k - 1
        kp1 = k + 1
        jm1 = j - 1
        jp1 = j + 1
        im1 = i - 1
        ip1 = i + 1
        g(n) = g(n) &
& - tmp1*(af+cf)    &
& - tmp2*(wrk(i,jm1,k)+wrk(i,jp1,k)) &
& - tmp3*(wrk(i,j,km1)+wrk(i,j,kp1))
         af = bf
         bf = cf
         cf = wrk(ip1+1,j,k)
    end do
  end do
    n = n + 1
end do
```

5.2.3.3 SIMD Processing

It is of great importance to effectively use SIMD extensions since (1) most modern computing platforms have incorporated SIMD capability in their processors, and (2) non-SIMD operations can severely reduce the floating-point performance. For example, BlueGene/P's PowerPC (PPC) 450 processor features a double precision, dual pipe, floating point unit (FPU). Dual FPU instructions operate on pairs of double-precision (DP) floating-point numbers, and perform SIMD processing to deliver two DP floating-point operations per cycle. If instructions are not SIMDized, attainable performance will be halved. Therefore, recent work has focused on exploiting SIMD capability of PPC 450 for petaflops applications on BlueGene/P [3].

In our earlier work, we used several code transformations for SIMDizing stencil computations and molecular dynamics simulations on Intel quadcore based platforms [25, 26, 78]. Here, we use Intel C compiler's SSE3 intrinsics to implement SIMD concepts for DFT application. Since the original DFT code is written in Fortran 90, we first port bottleneck subroutines to C language considering differences (e.g., array layout, variable pass mechanism, internal compiler naming convention) in two languages. Then we implement with SSE3 intrinsics to take advantage of XMM registers of Intel platform. Table 5.2 shows a sample code snippet.

Table 5.2: Original Fortran 90 code for linear mixing of wavefunctions subroutine and corresponding C variant written with explicit SSE3 intrinsics

Original implementation:

```
subroutine wfn_mix0(wfn1,coe2,wfn2)

use mod_mgdc
   implicit real*8 (a-h,o-z)
   dimension wfn1(mshnum), & & wfn2(mshnum)

   do i = 1, mshnum
       wfn1(i) = wfn1(i) + &
       & coe2*wfn2(i)
   end do

   return
end
```

Explicit SSE3 implementation:

```
void wfn_mix0_(double * wfn1,const double * coe2,const double * wfn2){
unsigned int i;
__m128d mmcoe2;
mmcoe2 = _mm_loaddup_pd(coe2);

for (i = 0; i < mshnum / 2; i++)
  _mm_store_pd( &wfn1[2*i],
  _mm_add_pd(_mm_load_pd(&wfn1[2*i]),
  _mm_mul_pd(_mm_load_pd(&wfn2[2*i]),mmcoe2)));

if ((mshnum % 2) != 0) {
for ( i = mshnum - mshnum % 2; i < mshnum; i++)
   wfn1[i] += (*coe2) * wfn2[i];}

return;}
```

In Table 5.2, first the `_mm_loaddup_pd` intrinsic loads the double-precision floating-point coefficient `coe2` into the lower and upper halves of the 16-byte long XMM register. Second, 2 double precision (8 byte) numbers, which are consecutive on 16-byte boundary aligned double-precision floating-point arrays, `wfn1` and `wfn2`, are loaded to another register by `_mm_load_pd` intrinsic. Finally, SIMD multiply-add operations are performed on XMM registers and results are stored back to computer memory. We will consider the memory latency hiding benefit of this implementation in the next section.

5.2.3.4 Memory Latency Hiding

Memory structure is critical to achieving high performance on recent processors featuring complex memory subsystems including registers, multilevel caches and storage buffers. Performance can be easily limited by the speed at which data can be transferred from system memory to processor and thus researchers recently developed several memory optimization frameworks for emerging architectures [4, 28, 54].

To hide memory latency, we treat memory alignment carefully. In Table 5.2, it should be noted that memory load/store operations are handled through `_mm_store_pd` and `_mm_load_pd` intrinsics, which are equivalent to `MOVAPD` in x86 assembly programming language. The use of `MOVAPD` to load an XMM register from a 128-bit memory location (or to store the contents of an XMM register into a 128-bit memory location) requires the operands to be aligned on a 16-byte boundary. We ensure 16-byte boundary alignment at application level, using the keyword `__attribute__((aligned(16)))`. This avoids redundant alignment checks during run time.

Similar to x86 microprocessor, the BlueGene/P architecture allows for two double-precision values to be loaded in parallel in a single cycle, provided that the load address is aligned such that the values loaded do not cross a cache-line boundary (which is 32-bytes). If a non-aligned data is accessed or modified at application level, the PowerPC hardware generates an alignment exception and the data is artificially aligned at the kernel level. Penalty for such context switch is in the order of thousands of cycles. Thus for efficient data transfer, data must be word aligned and must fit in a quadword aligned quadword. IBM XL Fortran compiler provides user level functions as compiler hints to inform the compiler that the incoming data is aligned according to specific byte boundary, so it can efficiently generate load and stores without alignment checks. At BlueGene/P platform, we ensure proper alignment at the application level and provide alignment guarantees to the compiler (using Fortran's `ALIGNX` function) for efficient load/store instruction generation.

5.2.3.5 Register and Cache Blocking

In section 3, we used blocking techniques targeting different levels of memory hierarchy: TLB, last level cache (LLC), and SIMD register file to increase memory performance on Intel quadcore based platforms [26]. Indeed these techniques has proven successful, as a consequence there are recent efforts to integrate such optimizations into performance tuning frameworks [15, 42].

In our DFT code, all of the top performance hotspot subroutines listed in section 5.2.1 feature large loops sweep the entire grid for wave functions. To increase spatial locality and reduce effective memory access time of the DFT application, we have implemented register and cache blocking methods mentioned in chapter 3.

In summary, the tangible benefit of our systematic optimization has been achieving ~15% performance improvement for the DFT application. Such an improvement for a production level application such as our DFT program can potentially save one-to-two orders of magnitude of petaflops•days of computing resources for a typical run or can extend the time and length scale of real-life MD simulations at the same computational cost. As the width of the SIMD register files increase in the emerging processors (e.g., Intel Sandy Bridge, Intel Ivy Bridge), and with the introduction of three-operand SIMD instruction format (e.g., advanced vector extensions to x86 instruction set), we expect our optimization techniques to be increasingly more effective on future computers.

Chapter 6

Conclusion

The focus of our work has been efficient parallelization and optimization of scientific computing applications on emerging parallel computer architectures. As high-performance computing systems feature increasingly complex levels of parallel computing hierarchy, it is imperative that application scientists overcome challenges involved with using such systems. Our contributions to this challenge, as we summarize in this chapter, enable effective use of emerging parallel computers to tackle vastly larger computational problems. In this chapter, we also detail ideas for future work that will make high performance computing systems play an even more important role in scientific discoveries.

6.1 Summary of Contributions

Productivity Tools on Emerging Architectures: We devised an efficient performance profiling methodology that targets hybrid computer systems. Based on Cell

Messaging Layer communication library, we developed the first cluster level performance profiler for the Roadrunner, which is the first hybrid supercomputer and the first supercomputer to attain a sustained petaflop/second performance. We demonstrated that our performance tool utilizes the Cell processor's high-bandwidth on-chip communication bus effectively to overlap collection of performance events with actual program execution. In addition, we showed that our library design maintains a very low memory footprint while it fully utilizes the exotic features of Cell architecture such as the coherent direct-memory-access mechanism. To demonstrate the utility of our tool, we ported several scientific applications to Cell platform. Experiments validated that our productivity tool provides a key starting point on applications performance optimization on Cell-based computing platforms.

Parallelization and Optimization Framework: We have presented parallelization algorithms and optimization strategies that allow leveraging multicore computer systems optimally for large-scale simulations. In particular, we have studied high-order stencil computations, which is a computation-bound mathematical kernel at the heart of most finite-difference time-domain (FDTD) method based simulations. We have developed a multilevel optimization framework for high-order stencil computations that combines: (1) data locality optimizations through auto-tuned tiling for efficient use of hierarchical memory; (2) register blocking and data parallelism via single-instruction multiple-data techniques to utilize registers and exploit data locality; (3) inter-core parallelization via multithreading and explicit non-uniform memory access control; and (4) inter-node parallelization via spatial decomposition. We have applied our optimization scheme to a 6th-order stencil based FDTD code. Our benchmarks on 32,768 BlueGene/P processors achieved over 98% weak-scaling parallel efficiency. We have also observed superlinear strong scaling on a large-cache x86 architecture based cluster. We have showed that our optimizations increase cache performance considerably. Our experiments have achieved the highest reported percentage peak performance for this computation pattern on general-purpose x86 architectures.

In a further effort to design scalable parallelization frameworks that would be applicable to the future generation of multipetaflops supercomputers, we have outlined a *metascalable* parallelization framework for atomistic simulations. In this study, we combined (1) an embedded divide-and-conquer (EDC) algorithmic framework based on spatial locality to design linear-scaling algorithms for high complexity problems; and (2) a tunable hierarchical cellular decomposition (HCD) parallelization framework to map these O(N) algorithms onto a multicore cluster based on hybrid implementation combining message passing and critical section-free multithreading.

A Systematic Optimization Scheme: We have unified our experience in performance optimization to improve the productivity of a real-world density functional theory application. As the first step of our systematic optimization scheme, we comprehensively evaluated the system level performance of this large-scale legacy code on various high-end computing platforms. Following this scalability analysis, we adopted a *model-guided system-level optimization* approach where we provided a theoretical analysis of the total computational cost implied by the application and optimally tuned algorithmic decomposition parameters such as the domain size and the buffer depth. As the second step of our optimization scheme, we used a *profiling-guided node-level optimization* methodology to restructure our program and implement variety of processor and memory optimizations. Based on the insights gained through performance analysis, we developed code transformations that improve data locality, concurrency and the floating-point performance in the original application where it is not efficient to use an existing performance library—if one exists at the least. The flowchart in Fig. 6.1 summarizes our systematic optimization scheme.

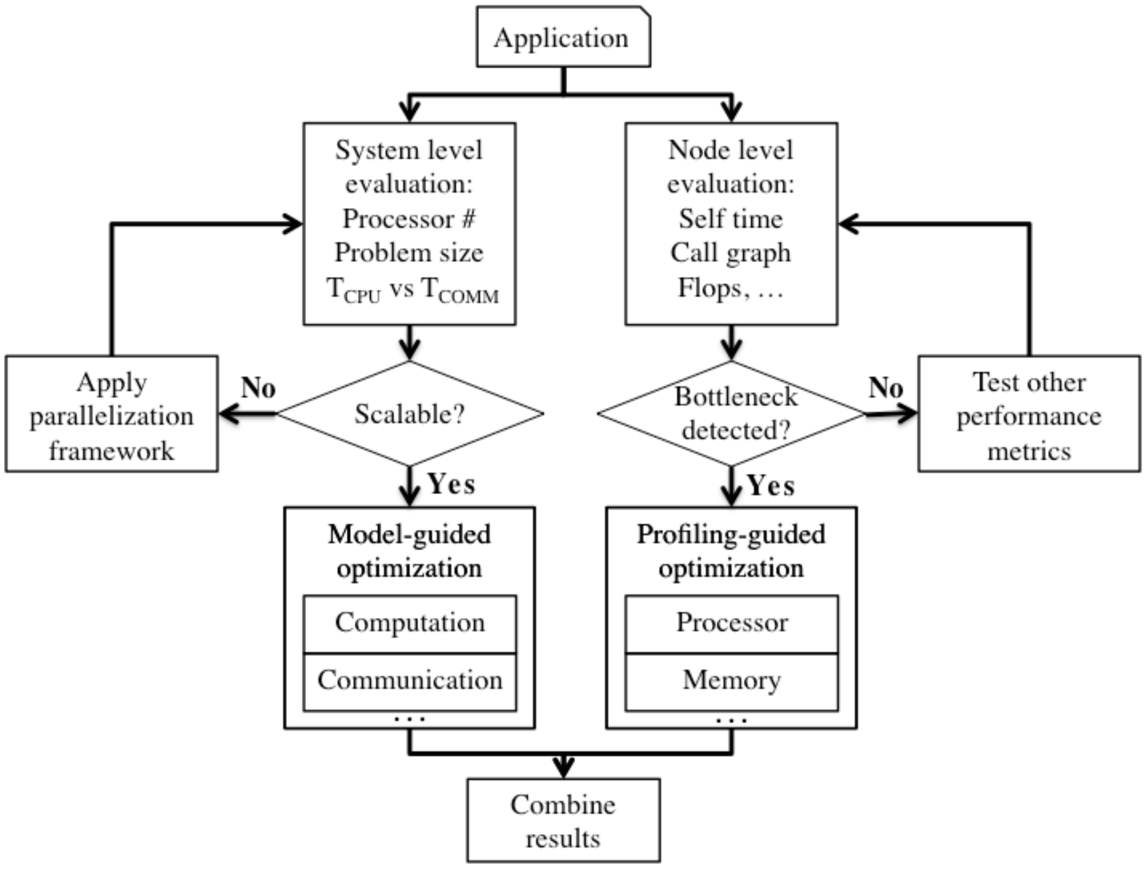

Application
System level evaluation: Processor # Problem size T_{CPU} vs T_{COMM}
Node level evaluation: Self time Call graph Flops, …
Apply parallelization framework
No
Scalable?
Bottleneck detected?
No
Test other performance metrics
Yes
Yes
Model-guided optimization
Computation
Communication
…
Profiling-guided optimization
Processor
Memory
…
Combine results

Figure 6.1: Systematic optimization process flow

To give a high-level description of our contributions, this thesis work develops a systematic end-to-end optimization approach for scientific applications on emerging high performance computing platforms. We underscore the significance of software productivity tools, in particular efficient performance analysis libraries, to achieve high application performance on new generation of supercomputers. To that end, we have developed a cluster level performance analysis tool on the first generation of hybrid supercomputers. On the application side, we effectively used such productivity tools to carefully port legacy code to new computing platforms and tune our algorithms to mimic architectural design decisions at the software level.

6.2 Future Work

As our study shows, it is imperative that application scientists are provided with a mature software stack to ensure the productivity of future computing systems. While we presented various memory efficient algorithms and processor optimizations in this dissertation research, it is still possible to extend this work to adapt new architectures beyond the ones discussed here.

As higher performance computer systems emerge, the extent of scientific computing is rapidly increasing. We believe our metascalable design paradigm will allow future applications continue to scale efficiently on the new generation of multipetaflops computers.

Bibliography

[1] F. F. Abraham, R. Walkup, H. Gao, M. Duchaineau, T. D. D. L. Rubia, and M. Seager, "Simulating materials failure by using up to one billion atoms and the world's fastest computer: Work-hardening," *Proceedings of the National Academy of Sciences of the United States of America,* vol. 99, pp. 5783-5787, 2002.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Pishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," 2006.

[3] N. Attig, "Computational physics with petaflops computers," *Computer Physics Communications,,* vol. 180, pp. 555-558, 2008.

[4] L. Bachega, S. Chatterjee, K. A. Dockser, J. A. Gunnels, M. Gupta, F. G. Gustavson, C. A. Lapkowski, G. K. Liu, M. P. Mendell, C. D. Wait, and T. J. C. Ward, "A high-performance SIMD floating point unit for BlueGene/L: Architecture, compilation, and algorithm design," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*: IEEE Computer Society, 2004.

[5] D. Bailey, J. Barton, T. Laninski, and H. Simon, "The NAS Parallel Benchmarks," NASA Ames Research Center, Moffett Field, California, 1991.

[6] B. Bansal, U. Catalyurek, J. Chame, C. Chen, E. Deelman, Y. Gil, M. Hall, V. Kumar, T. Kurc, K. Lerman, A. Nakano, Y. L. Nelson, J. Saltz, A. Sharma, and P. Vashishta, "Intelligent optimization of parallel and distributed applications," in *Proceedings of the Next Generation Software Workshop, International Parallel and Distributed Processing Symposium*: IEEE, 2007.

[7] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of Roadrunner," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* Austin, Texas: IEEE Press, 2008.

[8] R. Bordawekar, A. Choudhary, and J. Ramanujam, "Automatic optimization of communication in compiling out-of-core stencil codes," in *Proceedings of the 10th International Conference on Supercomputing* Philadelphia, Pennsylvania: ACM, 1996.

[9] M. Bromley, S. Heller, T. McNerney, and G. L. Steele, "Fortran at ten gigaflops: the connection machine convolution compiler," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* Toronto, Ontario, Canada: ACM, 1991.

[10] H. Brunst and W. E. Nagel, "Scalable performance analysis of parallel systems: Concepts and experiences," in *Parallel Computing: Software, Algorithms, Architectures Applications*, pp. 737-744, 2003.

[11] A. Buttari, J. Dongarra, and J. Kurzak, "Limitations of the PlayStation 3 for high performance cluster computing," University of Tennessee Computer Science, Tech. rep. 2007.

[12] R. Car and M. Parrinello, "Unified approach for molecular dynamics and density-functional theory," *Physical Review Letters,* vol. 55, pp. 2471-2474, 1985.

[13] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*: IEEE, 2007.

[14] J. R. Chelikowsky, Y. Saad, S. Ogut, I. Vasiliev, and A. Stathopoulos, "Electronic structure methods for predicting the properties of materials: Grids in space," *Physica Status Solidi (b),* vol. 217, pp. 173-195, 2000.

[15] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," USC Computer Science Technical Report June 2008.

[16] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: a performance view," *IBM J. Res. Dev.,* vol. 51, pp. 559-572, 2007.

[17] Y. C. Chen, Z. Lu, K. Nomura, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, "Interaction of voids and nanoductility in silica glass," *Physical Review Letters,* vol. 99, p. 155506, 2007.

[18] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review (SIREV),* vol. 51, pp. 129-159, 2009.

[19] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* Austin, Texas: IEEE Press, 2008.

[20] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New challenges in dynamic load balancing," *Applied Numerical Mathematics,* vol. 52, pp. 133-152, 2005.

[21] S. L. Dixon and K. M. Merz, "Fast, accurate semiempirical molecular orbital calculations for macromolecules," *Journal of Chemical Physics,* vol. 107, p. 879, 1997.

[22] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy, "The impact of multicore on computational science software," *CTWatch Quarterly,* vol. 3, pp. 11-17, 2007.

[23] H. Dursun, K. J. Barker, D. J. Kerbyson, and S. Pakin, "Application profiling on Cell-based clusters," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing* Rome, Italy: IEEE Computer Society, 2009.

[24] H. Dursun, K. J. Barker, D. J. Kerbyson, S. Pakin, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta, "An MPI performance monitoring interface for cell based compute nodes," *Parallel Processing Letters,* vol. 19, pp. 535-552, 2009.

[25] H. Dursun, K. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, "A multilevel parallelization framework for high-order stencil computations," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing* Delft, The Netherlands: Springer-Verlag, 2009.

[26] H. Dursun, K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta, "In-core optimization of high-order stencil computations," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, Nevada, 2009.

[27] S. Emmott and S. Rison, "Towards 2020 Science," Microsoft Research, Cambridge, UK 2006.

[28] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* Tampa, Florida: ACM, 2006.

[29] J. L. Fattebert and F. Gygi, "Linear scaling first-principles molecular dynamics with controlled accuracy," *Computer Physics Communications,* vol. 162, pp. 24-36, 2004.

[30] J. L. Fattebert and J. Bernholc, "Towards grid-based O(N) density-functional theory methods: optimized nonorthogonal orbitals and multigrid acceleration," *Physical Review B,* vol. 62, pp. 1713-1722, 2000.

[31] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. J. C. Ward, M. Giampapa, Y. Zhestkov, M. C. Pitman, F. Suits, A. Grossfield, J. Pitera, W. Swope, R. Zhou, S. Feller, and R. S. Germain, " Blue Matter: strong scaling of molecular dynamics on BlueGene/L " *Lecture Notes in Computer Science,* vol. 3992, pp. 846-854, 2006.

[32] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* Cambridge, Massachusetts: ACM, 2005.

[33] T. C. Germann and P. S. Lomdahl, "Recent advances in large-scale atomistic materials simulations," *Computing in Science and Engineering,* vol. 1, pp. 10-11, 1999.

[34] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz, "Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* Reno, Nevada: ACM, 2007.

[35] S. Goedecker, "Linear scaling electronic structure methods," *Rev. Mod. Phys.,* vol. 71, pp. 1085-1123, 1999.

[36] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics,* vol. 73, pp. 325-348, 1987.

[37] M. Gschwind, F. Gustavson, and J. F. Prins, "High performance computing with the cell broadband engine," *Scientific Programming* vol. 17, pp. 1-2, 2009.

[38] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in Cell's multicore architecture," *IEEE Micro,* vol. 26, pp. 10-24, 2006.

[39] F. Gygi, E. Draeger, B. R. d. Supinski, R. K. Yates, F. Franchetti, S. Kral, J. Lorenz, C. W. Ueberhuber, J. A. Gunneis, and J. C. Sexton, "Large-scale first-principles molecular dynamics simulations on the BlueGene/L platform using the Qbox code," *Proceedings of Supercomputing 2005,* ACM, 2005.

[40] F. Gygi, R. K. Yates, J. Lorenz, E. W. Draeger, F. Franchetti, C. W. Ueberhuber, B. R. d. Supinski, S. Kral, J. A. Gunnels, and J. C. Sexton, "Large-scale first-principles molecular dynamics simulations on the BlueGene/L platform using the Qbox code," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*: IEEE Computer Society, 2005.

[41] D. Hackenberg, H. Brunst, and W. E. Nagel, "Event tracing and visualization for Cell Broadband Engine systems," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing* Las Palmas de Gran Canaria, Spain: Springer-Verlag, 2008.

[42] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, Rome, Italy, 2009.

[43] P. Hohenberg and W. Kohn, "Inhomogeneous Electron Gas," *Phys. Rev.,* vol. 136, pp. B864-B871, 1964.

[44] HPCC/USC, http://www.usc.edu/hpcc.

[45] IBM, "IBM system Blue Gene solution: Blue Gene/P application development," IBM, 2008. http://www.redbooks.ibm.com/redbooks/pdfs/sg247287.pdf.

[46] IBM, *Software Development Kit for Multicore Acceleration Version 3.0*: IBM, 2007. http://www.ibm.com/developerworks/power/cell/.

[47] D. J. Kerbyson and K. J. Barker, "Automatic identification of application communication patterns via templates," in *18th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada, pp. 114-121, 2005.

[48] H. Kikuchi, R. K. Kalia, A. Nakano, P. Vashishta, H. Iyetomi, S. Ogata, T. Kouno, F. Shimojo, K. Tsuruta, and S. Saini, "Collaborative simulation grid: multiscale quantum-mechanical/classical atomistic simulations on distributed PC clusters in the US and Japan," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* Baltimore, Maryland: IEEE Computer Society Press, 2002.

[49] W. Kohn, "Density functional and density matrix method scaling linearly with the number of atoms," *Physical Review Letters,* vol. 76, pp. 3168-3171, 1996.

[50] W. Kohn and L. J. Sham, "Self-consistent equations including exchange and correlation effects," *Phys. Rev.,* vol. 140, pp. A1133-A1138, 1965.

[51] W. Kohn and P. Vashishta, in *Theory of the Inhomogeneous Electron Gas*, N. H. March and S. Lundqvist, Eds. New York: Plenum, pp. 79-184, 1983.

[52] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *J. Comput. Phys.,* vol. 229, pp. 7692-7714, 2010.

[53] M. S. Lam and M. E. Wolf, "A data locality optimizing algorithm," *ACM SIGPLAN Notices,* vol. 39, pp. 442-459, 2004.

[54] M. Liu, W. Ji, Z. Wang, J. Li, and X. Pu, "High Performance Memory Management for a Multi-core Architecture," in *IEEE International Conference on Computer and Information Technology*, Xiamen, China, pp. 63-68, 2009.

[55] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *International Journal of Parallel Programming,* vol. 29, pp. 217-247, 2001.

[56] S. Mintchev and V. Getov, "PMPI: High-Level Message Passing in Fortran 77 and C," in *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*: Springer-Verlag, 1997.

[57] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the Hilbert space-filling curve," *IEEE Transactions on Knowledge and Data Engineering,* vol. 13, pp. 124-141, 2001.

[58] A. Nakano, "A space-time-ensemble parallel nudged elastic band algorithm for molecular kinetics simulation," *Computer Physics Communications,* vol. 178, pp. 280-289, 2008.

[59] A. Nakano, "Multiresolution load balancing in curved space: the wavelet representation," *Concurrency: Practice and Experience,* vol. 11, pp. 343-353, 1999.

[60] A. Nakano, "Parallel multilevel preconditioned conjugate-gradient approach to variable-charge molecular dynamics," *Computer Physics Communications,* vol. 104, pp. 59-69, 1997.

[61] A. Nakano and T. J. Campbell, "An adaptive curvilinear-coordinate approach to dynamic load balancing of parallel multiresolution molecular dynamics," *Parallel Computing,* vol. 23, pp. 1461-1478, 1997.

[62] A. Nakano, R. K. Kalia, K. Nomura, A. Sharma, P. Vashishta, F. Shimojo, A. C. T. V. Duin, W. A. Goddard, R. Biswas, and D. Srivastava, "A divide-and-conquer/cellular-decomposition framework for million-to-billion atom simulations of chemical reactions," *Computational Materials Science,* vol. 38, pp. 642-652, 2007.

[63] A. Nakano, R. K. Kalia, K. Nomura, A. Sharma, P. Vashishta, F. Shimojo, A. C. T. v. Duin, I. W. A. Goddard, R. Biswas, D. Srivastava, and L. H. Yang, "De novo ultrascale atomistic simulations on high-end parallel supercomputers," *International Journal of High Performance Computing Applications,* vol. 22, pp. 113-128, 2008.

[64] A. Nakano, R. K. Kalia, and P. Vashishta, "Multiresolution molecular-dynamics algorithm for realistic materials modeling on parallel computers," *Computer Physics Communications,* vol. 83, pp. 197-214, 1994.

[65] A. Nakano, R. K. Kalia, P. Vashishta, T. J. Campbell, S. Ogata, F. Shimojo, and S. Saini, "Scalable atomistic simulation algorithms for materials research," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* Denver, Colorado: ACM/IEEE, 2001.

[66] A. Nakano, R. K. Kalia, P. Vashishta, T. J. Campbell, S. Ogata, F. Shimojo, and S. Saini, "Scalable atomistic simulation algorithms for materials research," *Sci. Program.,* vol. 10, pp. 263-270, 2002.

[67] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *Proceedings of the 2010 ACM/IEEE Conference on Supercomputing* New Orleans, Louisiana: IEEE Computer Society, 2010.

[68] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: a portable 'shared-memory' programming model for distributed memory computers," in *Proceedings of the 1994 Conference on Supercomputing* Washington, D.C.: IEEE Computer Society Press, 1994.

[69] K. Nomura, R. K. Kalia, A. Nakano, and P. Vashishta, "A scalable parallel algorithm for large-scale reactive force-field molecular dynamics simulations," *Computer Physics Communications,* vol. 178, pp. 73-87, 2008.

[70] K. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. C. T. van Duin, and W. A. Goddard, "Dynamic transition in the structure of an energetic crystal during chemical reactions at shock front prior to detonation," *Physical Review Letters,* vol. 99, p. 148303, 2007.

[71] K. Nomura, S. W. d. Leeuw, R. K. Kalia, A. Nakano, L. Peng, R. Seymour, L. H. Yang, and P. Vashishta, "Parallel lattice Boltzman flow simulation on a low-cost Playstation 3 cluster," *International Journal of Computer Science,* 2008.

[72] K. Nomura, R. Seymour, W. Wang, H. Dursun, R. K. Kalia, A. Nakano, P. Vashishta, F. Shimojo, and L. H. Yang, "A metascalable computing framework for large spatiotemporal-scale atomistic simulations," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing* Rome, Italy: IEEE Computer Society, 2009.

[73] S. Ogata, T. J. Campbell, R. K. Kalia, A. Nakano, P. Vashishta, and S. Vemparala, "Scalable and portable implementation of the fast multipole method on parallel computers," *Computer Physics Communications,* vol. 153, pp. 445-461, 2003.

[74] S. Pakin, "Receiver-initiated message passing over RDMA networks," in *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS)* Miami, Florida, 2008.

[75] "PARKBENCH: PARallel Kernels and BENCHmarks," Available from http://www.netlib.org/parkbench.

[76] M. Parker, S. Ketcham, and H. Cudney, "Acoustic wave propagation in urban environments," in *Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference*: IEEE Computer Society, 2007.

[77] L. Peng, M. Kunaseth, H. Dursun, K. Nomura, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, "Exploiting hierarchical parallelisms for molecular dynamics simulation on multicore clusters," *The Journal of Supercomputing,* vol. 57, pp. 20-33, 2011.

[78] L. Peng, M. Kunaseth, H. Dursun, K. Nomura, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta:, "A scalable hierarchical parallelization framework for molecular dynamics simulation on multicore clusters," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, Nevada, pp. 97-103, 2009.

[79] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kal, "NAMD: biomolecular simulation on thousands of processors," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* Baltimore, Maryland: IEEE Computer Society Press, 2002.

[80] S. J. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.,* vol. 117, p. 1, 1995.

[81] E. Prodan and W. Kohn, "Nearsightedness of electronic matter," *Proceedings of the National Academy of Sciences of the United States of America,* vol. 102, pp. 11635-11638, 2005.

[82] J. Ramanujam, S. Krishnamurthy, J. Hong, and M. Kandemir, "Address code and arithmetic optimizations for embedded systems," in *Proceedings of the 2002 Conference on Asia South Pacific Design Automation/VLSI Design*: IEEE Computer Society, 2002.

[83] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. V. Rajopadhye, "Towards optimal multi-level tiling for stencil computations," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium* Long Beach, California, 2007.

[84] G. Rivera and C. W. Tseng, "Tiling optimizations for 3D scientific computations," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing* Dallas, Texas: IEEE Computer Society, 2000.

[85] G. Roth, J. M. Crummey, K. Kennedy, and R. G. Brickner, "Compiling stencils in high performance Fortran," in *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing* San Jose, California: ACM, 1997.

[86] J. C. Sancho and D. J. Kerbyson, "Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE," in *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS)* Miami, Florida, 2008.

[87] D. E. Shaw, "A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions," *Journal of Computational Chemistry,* vol. 26, pp. 1318-1328, 2005.

[88] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossvary, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," *SIGARCH Comput. Archit. News,* vol. 35, pp. 1-12, 2007.

[89] G. Shen and A. C. Cangellaris, "A new FDTD stencil for reduced numerical anisotropy in the computer modeling of wave phenomena: Research Articles," *International Journal of RF and Microwave Computer-Aided Engineering,* vol. 17, pp. 447-454, 2007.

[90] F. Shimojo, R. K. Kalia, A. Nakano, and P. Vashishta, "Divide-and-conquer density functional theory on hierarchical real-space grids: parallel implementation and applications," *Physical Review B,* vol. 77, p. 085103, 2008.

[91] F. Shimojo, R. K. Kalia, A. Nakano, and P. Vashishta, "Embedded divide-and-conquer algorithm on hierarchical real-space grids: parallel molecular dynamics simulation based on linear-scaling density functional theory," *Computer Physics Communications,* vol. 167, pp. 151-164, 2005.

[92] M. Snir, "A note on N-body computations with cutoffs," *Theory of Computing Systems,* vol. 37, pp. 295-318, 2004.

[93] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference* vol. 1: MIT Press, 1998.

[94] A. Stathopoulos, S. Öğüt, Y. Saad, J. R. Chelikowsky, and H. Kim, "Parallel methods and tools for predicting material properties," *Computing in Science and Engineering,* vol. 2, pp. 19-32, 2000.

[95] A. Stathopoulos, Y. Saad, and K. S. Wu, "Dynamic thick restarting of the Davidson, and the implicitly restarted Arnoldi methods," *SIAM Journal on Scientific Computing,* vol. 19, pp. 227-245, 1998.

[96] M. M. Strout and P. D. Hovland, "Metrics and models for reordering transformations," in *Proceedings of the Workshop on Memory System Performance*: ACM, 2004.

[97] I. Szlufarska, A. Nakano, and P. Vashishta, "A crossover in the mechanical response of nanocrystalline ceramics," *Science,* vol. 309, pp. 911-914, 2005.

[98] Tianhe-1, National SuperComputer Center in Tianjin. http://www.top500.org/system/10186.

[99] TOP500List, http://www.top500.org.

[100] A. C. T. Van Duin, S. Dasgupta, F. Lorant, and W. A. Goddard, "ReaxFF: a reactive force field for hydrocarbons," *Journal of Physical Chemistry A,* vol. 105, pp. 9396-9409, 2001.

[101] A. F. Voter, F. Montalenti, and T. C. Germann, "Extending the time scale in atomistic simulation of materials," *Annual Review of Materials Research,* vol. 32, pp. 321-346, 2002.

[102] R. D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Practice and Experience,* vol. 3, 1991.

[103] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice Boltzmann simulation optimization on leading multicore platforms," in *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS)* Miami, Florida, 2008.

[104] D. Wonnacott, "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," in *Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium* Cancun, Mexico, 2000.

[105] W. Yang, "Direct calculation of electron-density in density-functional theory," *Physical Review Letters,* vol. 66, pp. 1438-1441, 1991.

[106] W. Yang and T.-S. Lee, "A density-matrix divide-and-conquer approach for electronic structure calculations of large molecules," *Journal of Chemical Physics,* vol. 103, pp. 5674-5678, 1995.