ON THE DESIGN OF ARCHITECTURE-AWARE ALGORITHMS FOR EMERGING APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Seunghwa Kang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2011

ON THE DESIGN OF ARCHITECTURE-AWARE ALGORITHMS FOR EMERGING APPLICATIONS

To my parents.

ACKNOWLEDGEMENTS

First of all, I want to thank my academic advisor, Prof. David A. Bader. As David holds his main affiliation in CSE and I joined Georgia Tech as an ECE student, I was extremely lucky to meet David. David has been an excellent advisor. I could work for the right project in the right timing under his guidance. He was also very supportive and understanding, and I could focus on research to finish my PhD. Thank you very much David!

I also want to thank Prof. Richard Vuduc. I could learn a lot during collaboration with him, and he also served as my committee member. I gratefully acknowledge Prof. D. Scott Wills, Prof. George F. Riley, and Prof. Bo Hong. Prof. Wills and Prof. Riley have served as my reading committee member from the dissertation proposal exam. Dr. Hong traveled from Savannah to serve as my dissertation defense committee member.

My colleagues in the HPC lab has been very helpful and I want to thank them as well. Thank you very much Kamesh Madduri, Virat Agarwal, Aparna Chandramowlishwaran, Vipin Sachdeva, Manisha Gajbe, Amrita Mathuriya, David Ediger, Karl Jiang, Xing Liu, Pushkar Pande, Sainath Mallidi, Robert McColl, Ivan Walker, Zhaoming Yin, Prashant Gaurav, Vyomkesh Tripathi, Jason Riedy, and Henning Meyerhenke. Logan Moon has provided great technical support during my doctoral research, and I also want to gratefully acknowledge his efforts.

Last but not least, I want to thank my parents and my brother, Seung Yub. I could not even start my PhD without them, and I definitely could not finish my PhD without their support and encouragement. Thank you very much Mom, Dad, and Seung Yub!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This dissertation maps various kernels and applications to a spectrum of programming models and architectures and also presents architecture-aware algorithms for different systems. Built on top of our experiences, we aim to provide feedback to system software researchers and computer architects.

The kernels and applications discussed in this dissertation have widely varying computational characteristics. For example, we consider both dense numerical computations—which are floating point intensive and have predictable data access patterns—and sparse graph algorithms—which are highly irregular and require intensive data synchronization. This dissertation also covers emerging applications from image processing, complex network analysis, and computational biology. JPEG2000 is the successor of JPEG and uses a mix of kernels with different computational requirements. To analyze large scale complex networks, we often need to process very large data to extract a network of interest. The structure of the extracted network often challenges modern hierarchical memory subsystems as well. We parallelize the COGNAC software package which reconstructs a phylogenetic tree using gene order data. The application has parallelism in multiple levels, and the degree of parallelism in each level varies widely.

We map these problems to diverse multicore processors (the Intel Harpertown and Nehalem architectures, the AMD Barcelona architecture, and the Sun UltraSparc architecture) and manycore accelerators (such as the IBM Cell Broadband Engine and NVIDIA GPUs). We also use new programming models—such as Transactional Memory, MapReduce, and Intel TBB—to address the performance and productivity challenges in the problems.

Our experiences highlight the importance of mapping applications to appropriate programming models and architectures. We also find several limitations of current system software and architectures and directions to improve those. The discussion focuses on system software and architectural support for nested irregular parallelism, Transactional Memory, and hybrid data transfer mechanisms. We believe that the complexity of parallel programming can be significantly reduced via collaborative efforts among researchers and practitioners from different domains. This dissertation participates in the efforts by providing benchmarks and suggestions to improve system software and architectures.

CHAPTER I

INTRODUCTION

Parallel algorithm design had been a topic of interest for only a portion of the programmers and researchers in the computing area. A group of researchers—often referred as *theoreticians*—have worked on extracting parallelism from seemingly sequential problems and designing parallel algorithms assuming a certain theoretical model; only a limited number of those algorithms have been implemented on real systems. Another group of researchers in the supercomputing area worked on solving large-scale science problems—mainly by manipulating extremely large matrices and vectors—using very expensive supercomputers. These problems often have obvious parallelism originated from the huge dimension of matrices and vectors, but scaling such computations to a large number of processors is far from trivial due to communication bandwidth and latency issues. Addressing such a challenge have been a task for a small number of highly-experienced top-notch programmers.

Microprocessor $\simeq$ Multicore (or Manycore) processor

As a consequence,

Computing $\simeq$ Parallel computing

Power wall, memory wall, and ILP wall [11] are forcing a paradigm shift to multicore and manycore architectures, and the landscape of computing is rapidly changing with the advent of multicore processors. Multicore—and even manycore—processors are replacing old single-core processors in nearly every area of computing. Most single-core processors will retire in the near future, and most computing systems will become parallel computing systems. Chip makers released and are developing microprocessors with varying architectures to balance power efficiency, programmability, and chip development cost for their target application areas. These changes impose several new research challenges.

Traditional algorithm works focus on designing new algorithms with low asymptotic complexity for a well defined theoretical model. Yet, transforming such algorithms to real implementations is not always straightforward. Due to the recent architectural changes in multicore processors and manycore accelerators, it becomes even more challenging or nearly impossible in many cases. Different architectures have widely varying execution units and memory subsystems, and this forces algorithm designers to consider such architectural design trade-offs from the very beginning.

Parallel computers are becoming ubiquitous, and this calls for new applications and more programmers to exploit those machines. New applications impose different computing challenges to traditional supercomputing applications. Finding new applications and identifying major computing issues in those applications are important research topics.

The computing industry cannot rely on a small number of highly-experienced programmers any more. However, new architectures often sacrifice programmability in favor of power efficiency, and this further increases the programming complexity to exploit parallel computers. System software researchers and computer architects are struggling to solve these problems, but their efforts will not come to fruition without help form researchers in application and algorithm domains. Designing future system software and architectures necessitates the solid understanding of future applications, key algorithms for those applications, and the role of software tools and hardware support in addressing the programming and performance challenges in those applications. This feedback loop (see Figure 1) is crucially important.

However, most system software researchers and computer architects do not understand applications and algorithms very well, and most researchers in the application

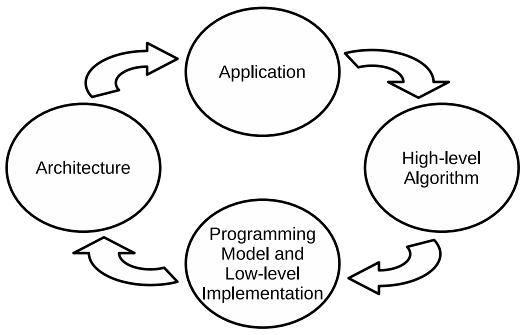


Figure 1: The feedback loop from emerging applications to future architectures.

and algorithm domains do not understand system software and computer architecture. System software researchers and computer architects design new tools and new microprocessors using old benchmarks. Researchers in the application and algorithm domains are significantly lagging behind the system- and architecture-level innovations. There have been several attempts to identify (*e.g.* [11, 19]) important kernels in current and future applications, but real applications require multiple kernels and mix those kernels in a complex way. Systems work well for individual kernels may not perform well for complete applications.

This dissertation presents parallel algorithms and performance tuning techniques for various applications and architectures (see Table 1). In presenting the algorithms and the techniques, we also discuss major computational challenges in those applications and the match between the algorithms for the applications and different programming models and architectures. Built on top of the lessons learned from the works, this dissertation provides feedback to system software researchers and computer architects.

1.1 The Organization of This Dissertation

The remainder of this dissertation is organized as the following. First, we discuss two classes of kernels with completely different computational characteristics in Chapters 2 and 3. Chapter 2 presents an inter-architectural comparison work for three

Table 1: Applications and architectures studied in this work.

area	**application/algorithm**	**architecture**
computational statistics	Pearson's and Kendall's methods (covariance computation)	Intel Harpertown, AMD Barcelona, IBM Cell Broadband Engine and PowerXCell 8i, and NVIDIA Tesla
graph algorithm	minimum spanning forest	Sun Niagara with software Transactional Memory
image processing	JPEG2000	IBM Cell Broadband Engine and AMD Barcelona
complex network analysis	sub-graph extraction and analysis	Sun Niagara and a MapReduce cluster with AMD Opteron processors
computational biology	phylogenetic tree reconstruction using gene order data	Intel Nehalem

kernels from computational statistics. These kernels are floating-point intensive and have a high degree of spatial locality and predictable data access patterns. The kernels also have large basic blocks (only a small number of branches) and do not require data synchronization. We study the impact of design trade-offs on the performance and programmability of the kernels for various multicore processors and accelerators—such as the Intel Harpertown and the AMD Barcelona architectures, the IBM Cell Broadband Engine and PowerXCell 8i architectures, and the NVIDIA Tesla architecture.

Chapter 3 maps irregular graph algorithms to Transactional Memory. Many graph algorithms are integer based and often have a very low degree of spatial and temporal locality. Data access patterns are highly irregular. Graph algorithms are often branchy and heavily involve data synchronization issues. We demonstrate that Transactional Memory can significantly reduce the programming and performance challenges in many graph problems in the chapter. We also present a new efficient Transactional Memory algorithm to compute a minimum spanning forest in irregular graphs. Our experimental results show the potential of Transactional Memory but also reveal the limitation of software Transactional Memory. We discuss the necessity of hardware support.

Chapters 4, 5, and 6 present parallel algorithm design and performance tuning works for three different emerging applications: JPEG2000, complex network analysis, and gene order data based phylogenetic tree reconstruction. Chapter 4 studies parallelization and performance tuning of JPEG2000 for the Cell Broadband Engine. JPEG2000 is the successor of widely used JPEG still image coding standard and consists of multiple kernels with different computational characteristics such as Embedded Block Coding with Optimized Truncation (EBCOT) and Discrete Wavelet transform (DWT). The EBCOT algorithm is irregular and branchy but has a fixed memory footprint. The DWT is communication-intensive and requires both horizontal and vertical scanning of the input image; this challenges current cache based memory subsystems. The Cell Broadband Engine provides a DMA based block data transfer mechanism, and we present performance tuning strategies to exploit the block data transfer mechanism and the processors's vector units.

Chapter 5 presents a hybrid system of a highly multithreaded architecture and a MapReduce cluster as a solution to address the computational challenges in complex network analysis problems which are both data- and communication-intensive. Analyzing complex networks involves managing very large data and traversing irregular graphs. The hybrid system of the two widely different architectures efficiently addresses the computational challenges. The result shows the importance of using right programming models and architectures for applications of interest.

We present parallelization of our *COGNAC* software package in Chapter 6. *COGNAC* reconstructs a phylogenetic tree using gene order data. *COGNAC* enables accurate reconstruction of a phylogenetic tree but is also computationally expensive. *COGNAC* has parallelism in multiple levels, and the degree of parallelism in each parallelization point varies widely for different input data sets and throughout computing phases.

Managing nested irregular parallelism is key to address the computational challenges in *COGNAC* using parallel computers. We use the combination of Intel TBB and microprocessors with simultaneous multithreading support to address the challenges with the minimum increase in programming complexity. The chapter highlights the importance of system software and architectural support in solving the parallel programming crisis.

Based on our experiences with different kernels, applications, programming models, and architectures, we discuss the major challenges in parallel algorithm design and implementation for modern parallel computers and provide feedback to system software researchers and computer architects in Chapter 7. Especially, we discuss issues related to system software support for nested irregular parallelism, Transactional Memory, and a hybrid data transfer mechanism. This dissertation is based on our published papers—such as [66], [64], [12], [63], and [65].

1.2 Research Contributions

The following summarizes the major contributions of this dissertation.

1. This dissertation discusses parallelization and performance tuning of kernels with widely varying computational characteristics and applications with a mix of kernels for a variety of architectures and programming models.
2. This dissertation presents mappings between classes of kernels and applications and different programming models and architectures.
3. This dissertation provides benchmarks for system software researchers and computer architects to test new programming models and architectural features.
4. This dissertation provides feedback to the system software and computer architecture communities from an application- and architecture-centric viewpoint.

CHAPTER II

UNDERSTANDING THE DESIGN TRADE-OFFS AMONG CURRENT MULTICORE SYSTEMS FOR NUMERICAL COMPUTATIONS

This chapter discusses the impact of architectural design trade-offs among various multicore processors and accelerators for dense numerical computations and is based on the materials in *Seunghwa Kang, David A. Bader, and Richard Vuduc, "Understanding the Design Trade-offs among Current Multicore Systems for Numerical Computing," The 23rd International Parallel and Distributed Processing Symposium (IPDPS), Rome, Italy, May 25th-29th, 2009.*

Multicore processors and accelerator architectures are replacing single-core processors in nearly every computing area. These systems are attractive to application developers because of their impressive peak computational potential and (in several cases) their energy-efficient processing capabilities. However, the architectures themselves are diverse and reflect a wide variety of design trade-offs. Consequently, we might expect the performance of a given application to be an even more sensitive function of the architecture than in previous generation single-core general purpose processors, which in turn is expected to affect software development costs significantly.

The research literature on software optimization for these multicore systems is growing rapidly, particularly for platforms based on the Cell Broadband Engine (Cell-/B.E.) [25, 62, 110] and for GPUs [100, 124]. Ryoo, *et al.* [108, 109] have published extensively on generalizing optimization principles for the NVIDIA GPUs based on the CUDA framework [96]. In addition to the optimization research for a single platform, several inter-architecture comparisons have been published as well. Williams, *et al.* [128] compared the performance of emerging multicore platforms, including the AMD dual-core Opteron processor, the Intel quad-core Harpertown processor, the Sun Niagara processor, and the Cell/B.E. processor for sparse matrix-vector multiplication. Also, there are several papers that compare the performance of CPUs, GPUs, FPGAs, the Cell/B.E. processor, and the Cray MTA-2 [82, 33, 25].

The focus of this chapter is on evaluating the impact of fundamental design trade-offs on a particular class of widely-used but simple-to-analyze software kernels. We study a range of systems with native double-precision support, described in detail in Section 2.1, including a two processor (2P) system with Intel quad-core Harpertown 2.5 GHz E5420 processors, a four processor (4P) system with AMD quad-core Barcelona 2.0 GHz 8350 processors, an IBM QS20 blade with two Cell/B.E. 3.2 GHz processors, an IBM QS22 blade with two PowerXCell 8i 3.2 GHz processors, and a desktop system equipped with one NVIDIA Tesla C1060 GPU. As the test systems have varying numbers of chips, clock frequencies, prices, and power consumptions, we focus on architectural design trade-offs and their impacts on different kernels rather than identifying the best performing processor.

We evaluate these systems experimentally using three kernels from computational statistics with differing computational characteristics. First, we create two kernels by extracting the most computationally intensive part of the covariance/correlation computation from the R statistics package [3]. These kernels are based on Pearson's method and Kendall's method [68], which we hereafter refer to as Kernel1 and Kernel2, respectively. The third kernel (Kernel3) is created by modifying Kernel2 to highlight each system's capability in processing highly floating-point intensive computation. Section 2.2 describes these kernels in more detail.

We also discuss our implementation and software optimization process, to highlight the challenges and complexities of software development for each architecture (Section 2.3). However, we consider the main contribution of this chapter to be our inter-architectural analysis, not the optimization work. Our experimental results highlight the performance of each system in executing a different mix of instructions for compute-bound and communication-intensive cases (Section 2.4). We consider both single-precision and double-precision performance for floating-point operations and aim to characterize the resulting performance in terms of each system's design choices.

2.1 *Inter-architectural Design Trade-offs*

This section describes five multicore systems of interest in this study, which are summarized in Table 2. In particular, each subsection considers a particular design dimension, and qualitatively summarizes the differences among the architectures. For additional processor details, we refer interested readers elsewhere [48, 34, 75].

Table 2: Summary of the test systems.

System	2P Harpertown E5420	4P Barcelona 8350	QS20-Cell/B.E.	QS22-PowerXCell 8i	Tesla C1060
Clock	2.5 GHz	2.0 GHz	3.2 GHz	3.2 GHz	1.296 GHz
Num. chips	2	4	2	2	1
Num. cores / chip	4	4	1 PPE + 8 SPEs	1 PPE + 8 SPEs	30 SMs × (8 single-precision SPs + 1 double-precision SP + 2 SFUs)
DP Gflop/s	80	128	29.2	204.8	78
SP Gflop/s	160	256	409.6	409.6	933
On-chip memory	6 + 6 MB L2 cache per chip	512 KB L2 cache per core and 2 MB shared cache per chip	256 KB local store per SPE	256 KB local store per SPE	16 KB shared memory per SM
DRAM type	DDR2	DDR2	Rambus XDR	DDR2	GDDR3
Shared DRAM access	UMA	NUMA	NUMA	NUMA	N/A
Latency hiding	cache + prefetching	cache + prefetching	double (or triple) buffering	double (or triple) buffering	hardware-multithreading
Theoretical peak bandwidth (GB/s)	21.4	42.8	51.2	51.2	102
Power (W)	2 × 80 (per chip)	4 × 75 (per chip)	315 (per blade)	250 (per blade)	200 (max., per board)
Compiler	Intel icc (10.1.018)	Intel icc (10.1.015)	IBM xlc (10.1)	IBM xlc (10.1)	NVIDIA nvcc (release 2.0 V0.2.1221)
Optimization flag	-fast	-fast	-O5 -qarch=cellspu -qtune=cellspu	-O5 -qarch=edp -qtune=edp	-O3 -arch sm_13

Among "conventional" general purpose multicore microprocessors, we consider the Intel quad-core E5420 Harpertown processor and AMD's quad-core 8350 Barcelona. The Intel Harpertown and AMD Barcelona processors share a similar micro-architecture, but take distinct approaches to cache hierarchy and memory subsystem design. The Intel Harpertown has 12 MB L2 cache memory, and two cores among four cores in a chip share 6 MB L2 cache. The AMD Barcelona has dedicated 512 KB L2 cache per core in addition to 2 MB L3 cache shared by all four cores in a chip. Also, the AMD Barcelona supports the NUMA (Non-Uniform Memory Access) architecture while the Intel Harpertown is based on the UMA (Uniform Memory Access) architecture.

Among the multicore accelerator systems, we consider two generations of the STI Cell/B.E. processor and the NVIDIA Tesla C1060 GPU. The Cell/B.E. processor is a heterogeneous multicore processor with one conventional PowerPC core ("PPE") and eight specialized single-instruction multiple-data (SIMD) accelerators ("SPEs"). The Tesla C1060 is a GPU from NVIDIA and delivers 933 Gflop/s in single-precision with native 78 Gflop/s double-precision support. The C1060, based on the Tesla architecture [75], has 30 SMs (Streaming Multiprocessors), and each SM has 8 SPs (Streaming Processors) for single-precision and one SP for double-precision support. Also, each SM has two SFUs (Special Function Units) for transcendental functions and attribute interpolation.

2.1.1 Requirements for Parallelism

The Intel Harpertown, AMD Barcelona, and Cell/B.E. processors have four to nine cores, and each core supports SIMD instructions for acceleration. To exploit the parallelism in these processors, these chips require coarse-grain parallelism for their multiple cores in addition to SIMD parallelism.

In contrast, the Tesla C1060 has 240 SPs and 60 SFUs for single-precision. In addition, to be detailed in Section 2.1.4, the Tesla architecture adopts massive hardware multithreading to hide DRAM access latency. The Tesla C1060 requires at least several thousand-way parallelism to exploit its architectural features. Also, the Tesla C1060 requires SIMT (Single Instruction Multiple Threads) parallelism, and every thread in a single warp (a group of 32 threads) needs to agree on the execution path to maximize chip utilization.

2.1.2 Computation Units

Each core of the Intel Harpertown or AMD Barcelona processor can retire up to two SIMD floating-point instructions (one SIMD add and one SIMD multiply) [9, 58]; thus each core can deliver 4 double-precision floating-point operations per cycle. Also the Intel and AMD cores can execute integer instructions in parallel with floating-point instructions. Each SPE in the Cell/B.E. processor has two (even and odd) pipelines. The even pipeline can execute floating-point instructions, fixed point arithmetics, and logical and word-granularity shift and rotate instructions. The odd pipeline executes load/store instructions and fixed point byte-granularity shift, rotate mask, and shuffle instructions. Each SPE can retire one SIMD FMA (fused-multiply-and-add) instruction per cycle. Each SP in the Tesla C1060 executes one scalar instruction per cycle, including a single-precision FMA instruction. Each SFU can also execute four single-precision multiply instructions per cycle. One SP for double-precision support can retire one double-precision FMA instruction per cycle.

The sustainable flop-rate is highly affected by the mix of different instruction types in an execution stream and the structure of the computation unit. The Intel and AMD cores have separate multiply and add units instead of a FMA unit and also can run integer instructions in parallel; thus, they can more flexibly execute different combinations of integer and floating-point instructions. Still, to fully utilize both floating-point multiply unit and add unit, this chip requires a 1:1 ratio of multiply and add instructions. One SPE of the Cell/B.E. can issue only one floating-point instruction in a single cycle (whether it is FMA or not), so if multiply cannot be fused with add, the achievable peak flop-rate becomes halved. Still, the Cell/B.E. can run several types of fixed point instructions in parallel with floating-point operations. Fully exploiting the SMs in the Tesla C1060 requires FMA instructions and additional multiply instructions for the SFUs. Also, one SP can execute only one instruction per cycle.

2.1.3 Start-up Overhead

Kernel launching is faster on the Intel or AMD processors than on the Cell/B.E. or the Tesla C1060. Thus, for a small amount of computation, these general purpose processors outperform the accelerators while the accelerator architectures often exhibit impressive performance for larger data [82].

The test systems also incur different levels of data off-loading overhead. In the Harpertown, Barcelona, and Cell/B.E. processors, the off-loading overhead is largely determined by the off-chip memory bandwidth. In contrast, GPU systems incur additional host memory to on-board device memory data transfer via a slower PCI Express bus. While GPUs can partially hide the off-loading overhead with asynchronous data transfer (*i.e.*, double-buffering), this mechanism currently works only for page-locked memory and incurs additional programming overhead [98]. To amortize the off-loading overhead, GPUs require higher computational intensity than other processors [32, 114, 82]. However, the Tesla C1060's on-board memory is much larger (4 GB) than the Harpertown or Barcelona's cache memory (12 or 2 MB) or the Cell-/B.E.'s local store (256 KB per SPE $\times$ 8 SPEs). Accordingly, the Tesla C1060 can fit larger data into its on-board memory to minimize the data transfer over the PCI Express bus.

2.1.4 Memory Latency Hiding

The Intel Harpertown and AMD Barcelona processors hide memory latency via cache memory and prefetching mechanisms. The Cell/B.E. overlaps computation with communication via double- or triple-buffering. Double buffering efficiently hides the latency but requires explicit software intervention. The Tesla C1060 tolerates several hundred cycle DRAM access latency via massive hardware multithreading. The Tesla GPU also has per SM shared memory (16 KB) in addition to constant cache and texture cache. Yet, as each SM can run hundreds of threads in parallel, these on-chip memories have little performance impact if there is only a low degree of data sharing among different threads.

2.1.5 Control over On-chip Memory

For the cache-based multicore processors, cache memory is managed by hardware using the LRU (Least Recently Used) policy (or its variants), and programmers have essentially no control over cache partitioning. By contrast, programmers can explicitly manage on-chip ("local store") memory on the Cell/B.E. The Tesla C1060, along with the NVIDIA CUDA framework, also allows programmers to control the placement of data arrays to the chip's different types of memories.

2.1.6 Main Memory Access Mechanisms and Bandwidth Utilization

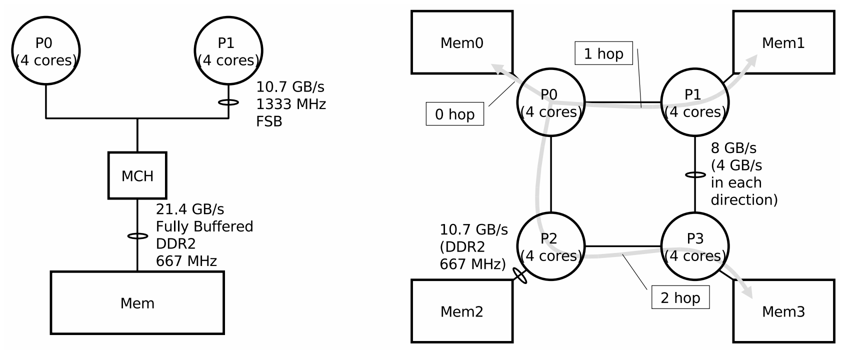


Figure 2: The 2P Intel Harpertown system with the UMA architecture (left) and the 4P AMD Barcelona system with the NUMA architecture (right).

The AMD Barcelona and Cell/B.E. processors use the NUMA (Non-Uniform Memory Access) architecture, while the Intel Harpertown processor adopts the UMA (Uniform Memory Access) architecture. UMA is conceptually simpler but NUMA has scalability advantages if cores running on different chips access distinct data arrays. In particular, by locating data to the chip's local main memory, we can minimize the contention and interference in the main memory interface. For instance, if a computation accesses read-only data multiple times, we can replicate the data to each processor's local DRAM to maximize the bandwidth utilization for accessing the data. The Tesla C1060's device memory does not support shared memory access over two or more GPUs.

The systems also differ in their deliverable memory bandwidth [102]. In terms of peak aggregate bandwidth, the 2P Harpertown system can deliver 21.4 GB/s, the 4P Barcelona system supports 42.8 GB/s, and the QS20 and QS22 blades support 51.2 GB/s for main memory access. The Tesla C1060 supports 102 GB/s peak bandwidth to its 4 GB device memory. However, there is often a significant gap between the peak bandwidth and the sustainable bandwidth [59, 94]. The gap is even larger for multicore processors due to the interference among multiple threads performing data accesses [93, 103, 105].

The systems adopt different memory controller architectures. In general, most memory controllers are designed to deliver the highest data transfer rate when accessing a large contiguous chunk of data, in particular by exploiting the maximum locality of a row buffer and bank level parallelism. Switching between DRAM reads and writes should also be minimized to achieve the highest bandwidth utilization. However, even for simple computations in which each thread is reading a linear array with stride one, memory requests coming from multiple cores can be intermixed, thereby destroying the locality and parallelism of a DRAM chip. For the Intel Harpertown and AMD Barcelona processors, the granularity of memory access is the lowest level cache line size (64 byte). For data-intensive applications, memory access requests from multiple cores with the size of 64 bytes can be heavily interleaved. The situation is even worse for the Intel Harpertown processor with its UMA configuration, as the memory controller hub must mix memory access requests coming out from two different chips. For NVIDIA GPUs, the access granularities are 32, 64, and 128 bytes [98].

By contrast, the Cell/B.E. adopts a different memory subsystem. First, each SPE generates DMA requests with significantly larger sizes—up to 16 KB. Even for communication-intensive applications, each SPE issues DMA requests in an intermittent fashion. This minimizes the inter-core interference in DRAM accesses. Therefore, programmers can maximize the bandwidth utilization by increasing the size of DMA accesses [62, 110]. Thus, the Cell/B.E. architecture fundamentally lends itself to higher bandwidth utilization than other systems, in spite of the significant effort toward increasing bandwidth utilization in general purpose multicore processors [93, 79, 59]. The AMD Barcelona processor adopts optimized scheduling algorithms especially for interleaved DRAM access streams as well [9]. Williams, *et al.*, also demonstrated the first-generation Cell/B.E.'s high bandwidth utilization [128], though an open question is the impact of adopting different DRAM technologies (*i.e.*, the first- and second-generation Cell/B.E. architectures adopt different DRAM technologies, namely, XDR and DDR2, respectively). Finally, the Cell/B.E. has an additional advantage in optimizing its memory controller, as this processor targets streaming applications that are highly latency tolerant. The Cell/B.E.'s memory controller can focus on bandwidth utilization, while general purpose multicore processors attempt to address the significantly more difficult problem of balancing bandwidth utilization with fairness and latency issues [94, 93, 103].

2.1.7 Ideal Software Implementations

To optimize the code for the Intel Harpertown, AMD Barcelona, and Cell/B.E. processors, one first needs to identify coarse-grain parallelism and partition data to exploit all the cores. One then needs to consider data layout for higher data transfer and vectorization efficiency. The Harpertown and Barcelona processors are significantly less sensitive to data alignment than the Cell/B.E., since they support multiple additional instructions for unaligned data accesses; however, data layout still affects the performance in a non-negligible amount. At high level, optimizing for the Cell/B.E. does not differ much from the Harpertown and Barcelona processors, but the actual implementation is significantly more complex as programmers need to explicitly program for data transfer within the local store size limit of 256 KB. In addition, the gap between the performance of baseline and optimized code is significantly higher for the Cell/B.E., and this often mandates manual optimization.

The optimization process for the Tesla C1060 is largely different from the above three processors. For the Tesla C1060, easily identifiable coarse-grain parallelism does not suffice to fully exploit the chip. Thus, the optimization should focus on extracting additional parallelism. To benefit from the high bandwidth and low latency on-chip memories, programmers need to modify an algorithm to maximize data sharing among multiple threads. Data coalescing and broadcasting mechanisms are also crucial to achieve high performance, and this also needs to be considered in algorithm design. For the Tesla C1060 or other CUDA enabled GPUs, the key challenge arises from high level algorithm design, and the actual implementation is less complex in terms of code size.

For the NUMA-based systems, one can gain significant speedup for bandwidth-intensive algorithms by controlling thread binding and data allocation. The optimization result for the Cell/B.E. often is more predictable than the x86 based architectures or the Tesla C1060 owing to its simpler architecture. For the x86 based architectures, the multi-level memory hierarchy with different latency, size, and associativity in each level and complex and adaptive prefetching mechanisms across the memory hierarchy significantly complicate performance analysis. The Tesla C1060 optimization is complicated by its large search space as well, which is non-linear in nature [109].

2.2 Kernel Descriptions and Qualitative Analysis

For our evaluation, we consider two versions of covariance computation based on Pearson's method and Kendall's method, as implemented in the open-source R statistics package [3]. We also create the third Kernel by modifying the second kernel based on Kendall's method. Given two test data sets, represented by an $n_X \times n$ matrix X, an $n_Y \times n$ matrix Y, and pre-computed mean vectors $\bar{x}$ and $\bar{y}$ of length n_X and n_Y, respectively, the basic covariance computation (based on Pearson's method) produces an $n_X \times n_Y$ matrix C such that

$$C_{ij} \leftarrow \frac{1}{n-1} \sum_{k=1}^{n} (X_{ik} - \bar{x}_i) \cdot (Y_{jk} - \bar{y}_j)$$

In this section, we describe the three kernels, and explain their high-level characteristics.

2.2.1 Conventional Sequential Code

Code 2.1 presents the C implementation of the basic covariance kernel based on Pearson's method. We refer to this code as "Kernel1." "Kernel2" computes covariance using Kendall's method, and we artificially create "Kernel3" by modifying Kernel2. Code 2.2 depicts Kernel2 and Kernel3. Kernel1 and Kernel2 are adopted from the R project source code [3].

In Kernel1, we can first subtract the mean vector from each matrix operand to remove the redundant subtracts. Then, transposing matrix Y converts this algorithm to a dense matrix multiplication problem, which is extensively studied and also there

```
//p_x: a pointer for X
//p_y: a pointer for Y
//p_xm: a pointer for $\bar{x}$
//p_ym: a pointer for $\bar{y}$
//p_ans: a pointer for $C$
for(i = 0 ; i < $n_X$ ; i++) {
  p_xx = &p_x[i * n];
  xxm = p_xm[i];
  for(j = 0 ; j < $n_Y$ ; j++) {
    p_yy = &p_y[j * n];
    yym = p_ym[j];
    sum = 0.0;
    for(k = 0 ; k < n ; k++) {
      sum += (p_xx[k] - xxm) * (p_yy[k] - yym);
    }
    p_ans[i * $n_Y$ + j] = sum / (n - 1);
  }
}
```

Code 2.1: C code for Kernel1

is a highly optimized BLAS library for the problem. Kernel2 and Kernel3 have more complex data access patterns but still can be optimized based on the cache blocking approach. Initially, *we intentionally ignore these particular optimization opportunities for the following two reasons.* First, we wish to stress the memory systems experimentally, and secondly, we want to show the more typical and intuitive optimization process that is common in practice. Then, if memory bandwidth turns out to be a performance bottleneck, we implement the blocking approach. We focus on highlighting the impact of architectural design trade-offs on performance and programmability. In particular, we do not intend to conclude which system is the "best" for computing covariance, nor do we claim to have implemented the best possible covariance code.

2.2.2 Basic Algorithmic Analysis

The memory footprint of all three kernels is $O((n_X + n_Y) \times n)$ and the size of two input matrices are typically much larger than mean vectors or the output matrix. The computational complexity is $O(n_X \times n_Y \times n)$ for Kernel1 and $O(n_X \times n_Y \times n^2)$ for Kernel2 and Kernel3. While these kernels are compute-intensive in their asymptotic

```
//p_x: a pointer for X
//p_y: a pointer for Y
//p_ans: a pointer for C
for(i = 0 ; i < n_X ; i++) {
  p_xx = &p_x[i * n];
  for(j = 0 ; j < n_Y ; j++) {
    p_yy = &p_y[j * n];
    sum = 0.0;
    for(k = 0 ; k < n ; k++) {
      for(n1 = 0 ; n1 < n ; n1++) {
#if SIGN//Kernel2
        sum += sign((p_xx[k] - p_xx[n1])
          * (p_yy[k] - p_yy[n1]));
#else//Kernel3
        sum += (p_xx[k] - p_xx[n1])
          * (p_yy[k] - p_yy[n1]));
#endif
      }
    }
    p_ans[i * n_Y + j] = sum;
  }
}
```

Code 2.2: C code for Kernel2 and Kernel3

notations, if the entire memory footprint does not fit into the on-chip memory of the test systems, then these kernels can be bandwidth-bound. All three kernels have obvious $n_X \times n_Y$ way parallelism as every pair of rows from matrix X and Y can be computed independently. Also, if we ignore the floating-point associativity issues, we can also trivially parallelize the innermost loop of Kernel1 and the second innermost loop of Kernel2 and Kernel3. For Kernel1, if we execute the code in a sequential way, there is higher temporal locality in the row data of matrix X than the row data of matrix Y. For Kernel2 and Kernel3, if we can place two rows from matrix X and Y on on-chip memory, we can perform $O(n^2)$ computation over $O(n)$ data without off-chip memory access. The total number of flops executed by Kernel1 is $(n_X + n_Y) \times n + n_X \times n_Y \times n \times 2$, and $n_X \times n_Y \times n^2 \times 4$ for Kernel2 and Kernel3.

2.3 Baseline Architecture-specific Implementations

For subsequent evaluation, we create a basic parallel implementation for each architecture, described in this section. These implementations include "baseline" parallelization and tuning, meaning they include some degree of platform-specific tuning but are not extensively tuned. Again, as Section 2.2.1 states, our focus is on system evaluation and not on kernel optimization.

2.3.1 Intel Harpertown (2P) and AMD Barcelona (4P) Multicore Implementations

We can easily parallelize the outermost loop of all three kernels with OpenMP or pthreads for our 8 and 16 core systems, assuming sufficiently large n_X (n_Y). For Kernel1, we apply auto-vectorization with two directives, *#pragma unroll(16)* and *#pragma vector aligned*, achieving comparable performance to an intrinsics-based vectorization approach.

For Kernel2, *sign()* function involves branches, lowering the performance significantly. We replace the branch with an SSE compare (*e.g.*, *_mm_cmpgt_pd()* and *_mm_cmplt_pd()*) and bitwise operations (*e.g.*, *_mm_and_pd()* and *_mm_or_pd()*). The Intel icc compiler have failed to perform this replacement automatically, and so we hand-code this translation to use SIMD intrinsics. The Kernel3 code can be trivially vectorized in the same way.

For the 4P Barcelona system, which is NUMA-based, our code replicates the input matrices to all four chips' local DRAM, and pins the threads to each core. The replication cost can be amortized with multiple reads, and this optimization maximizes the available bandwidth while minimizing the interference.

Even though our test kernels are asymptotically compute-intensive, if the input matrices do not fit into the on-chip cache memory, these algorithms can be bandwidth-bound. The blocking approach can reduce the amount of off-chip data transfer at the cost of increased implementation complexity. Also, for the Harpertown and Barcelona processors, selecting the optimal block size requires exhaustive search over parameter space as it is a complex function of the multiple levels of cache hierarchy and their size and associativity. This exhaustive search is beyond the scope of our work and we set the block size based on heuristics.

2.3.2 STI Cell/B.E. (2P) Implementation

The Cell/B.E. implementation resembles the 4P AMD Barcelona implementation, though the Cell/B.E. provides an additional opportunity for fine-tuning owing to the higher level of control over on-chip memory supported by the architecture. In particular, observe that a row data of matrix X has, assuming the given loop order, higher temporal locality than a row data of matrix Y. Thus, we can assign a larger buffer for matrix X than Y. Furthermore, to reduce the bandwidth requirement even when a single row does not fit into the local store, our code allocates additional small buffers for streaming. In this case, our code reads data from the larger buffer for matrix X and Y for accessing the initial part of the row (which fits into the local store), and then our code switches to the streaming mode with the smaller buffers for the remaining.

However, fine-grained control over on-chip memory significantly increases the coding complexity, especially when the on-chip memory requirement varies as a function of the input data size. The blocking approach, even though it adds additional complexity in high level, fixes the on-chip memory requirement regardless of the input data size. Accordingly, the blocking approach can reduce the coding complexity for the Cell/B.E. in addition to the improved performance. For the Cell/B.E., the impact of different block size is easier to understand owing to its simple memory subsystem. Larger block height reduces the amount of traffic whereas larger block width increases the iteration count of the innermost loop to improve the compute efficiency. We can also simply pick the largest block size that fits into the local store instead of considering different cache sizes in the memory hierarchy.

2.3.3 NVIDIA Tesla C1060 Implementation

For the NVIDIA Tesla C1060, the $n_X \times n_Y$-way parallelism may not be sufficient for practical data set sizes. Even when $n_X \times n_Y$ is very large, having every thread processes a distinct pair of rows can lead to poor bandwidth utilization (no coalescing in data transfer) or low on-chip cache utilization (no data sharing). For Kernel1, we partition the innermost loop with chunks of size 16 elements (a half warp, as high memory bandwidth utilization is achieved when the memory accesses from a half warp can be coalesced [98]). Each thread in a half warp processes one element out of 16 elements in a chunk to maximize the coalescing. For Kernel2 and Kernel3, we partition the second innermost loop identical to the case of Kernel1. In this case, every thread in a half warp traverses same row data in a synchronized way (in the innermost loop of Code 2.2, array index k remains constant and only array index *n1* changes. In our optimized code, every thread in a same half warp accesses *p_xx* and *p_yy* with same *n1* but different k), and we can use the on-chip shared memory to exploit this fact. As every thread accesses a same data element, we can use the shared memory's broadcasting mechanism as well.

One critical issue is *sign()* function, which involves branch instructions. The NVIDIA CUDA compiler replaces branch instructions with predicates when the number of instructions controlled by the branch is equal to or less than the threshold value (4 or 7 instructions) [98]. Therefore, by using the CUDA framework, we do not need to manually optimize for *sign()* function, as we do on the Intel Harpertown, AMD Barcelona, and Cell/B.E. platforms. Optimization for the Tesla C1060 is more involved in high level, but simpler to program than the Cell/B.E. for these kernels.

Also, the proper use of on-chip cache memory significantly reduces the bandwidth requirement, and Kernel2 and Kernel3 become compute-bound even without explicit blocking.

2.3.4 A Quantitative Comparison of Implementation Costs

Table 3: Quantitative comparison of implementation costs in terms of code size and implementation time. This excludes the code for kernel invocation and the residual part computation.

	kernel code size (# of lines)	approximate coding time
Harpertown and Barcelona - initial	335	1 day
Harpertown and Barcelona - blocking	419	2 days
Cell/B.E. - initial	1620	7 days
Cell/B.E. - blocking	1004	2 days
Tesla C1060 - initial	52 (Kernel1) + 97 (Kernel2/3)	2 days
Tesla C1060 - blocking	88 (Kernel1)	1 day

Table 3 summarizes the comparison. For the Cell/B.E., the blocking approach fixes the local store space requirement regardless of the input data size, and simplifies the coding in addition to the improved performance. We can also identify that the code size for the Tesla C1060 is significantly smaller than the other architectures. For the Tesla C1060, the challenge is in extracting additional parallelism and best exploiting the memory subsystem (based on the data access coalescing and broadcasting mechanisms and the efficient use of the shared memory).

2.4 Experimental Results

Recall the evaluation platforms from Table 2. To measure the sustained bandwidth of the 2P Harpertown system, we use PAPI [2] and count the number of memory bus transactions. For the 4P Barcelona system, we use AMD CodeAnalyst [1] and count the number of DRAM accesses. For the systems with the Cell/B.E. processors, we attach counter variables to every DMA memory requests and ignore the PPE initiated traffic. For the Tesla C1060, we estimate the total bandwidth requirement using the following equations: $n_X \times n_Y \times n \times$ sizeof(float or double) $\times 2$ for Kernel1 and $n_X \times n_Y \times n \times n \times$ sizeof(float or double) $\times 2 \times \frac{1}{16}$ (a half warp width, owing to data sharing) for Kernel2 and Kernel3. For the NUMA-based AMD Barcelona and Cell/B.E. architectures, our code replicates matrix X and Y, which are read multiple times, for higher bandwidth utilization. The reported numbers include this replication cost and the off-loading overhead to the device memory in the Tesla C1060.

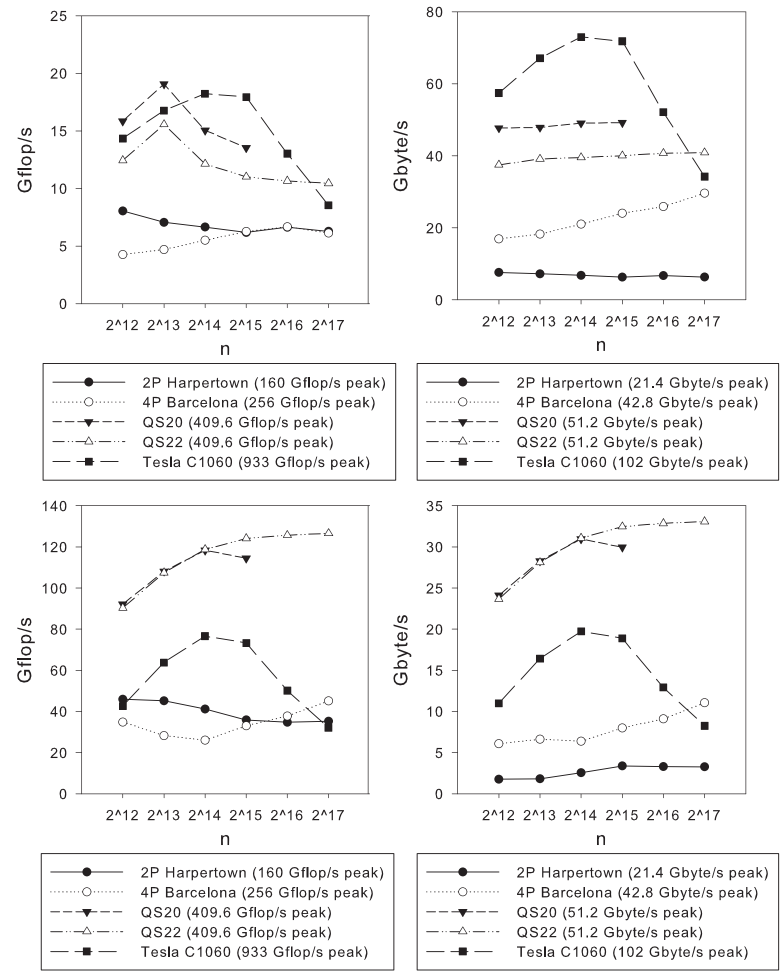


Figure 3: Sustained Gflop/s (left) and bandwidth utilization (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel1 (single-precision). Missing points for the QS20 are due to memory allocation failure. Here, $n_X = n_Y = 1024$.

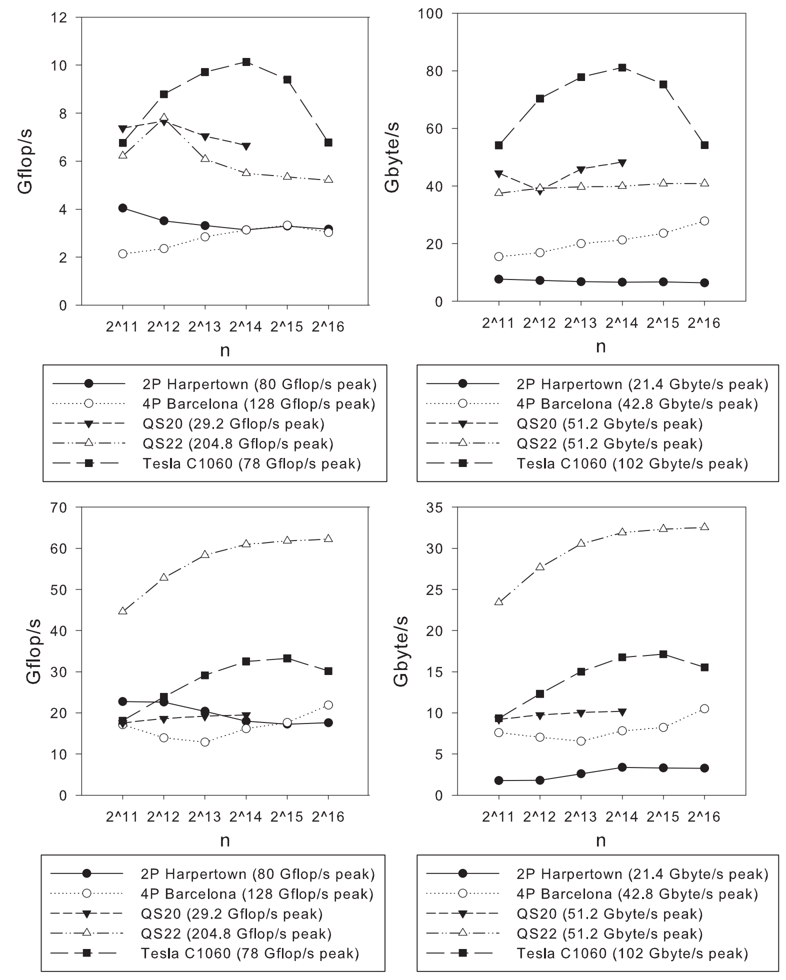


Figure 4: Sustained Gflop/s (left) and bandwidth utilization (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel1 (double-precision). Missing points for the QS20 are due to memory allocation failure. Here, $n_X = n_Y = 1024$.

2.4.1 Kernel1

The top half of Figure 3 depicts the sustained Gflop/s and bandwidth utilization for Kernel1 in single-precision with the initial implementation. Although the algorithm is computationally intensive, the performance is bounded by memory bandwidth since the entire data do not fit into the on-chip memory. The Tesla C1060 benefits from its high bandwidth to on-board DRAM, but the sustained bandwidth is lower than the theoretical peak and varies significantly for different input matrix sizes. The off-loading overhead accounts for 18% (for the smallest matrix) to 2.4% (for the largest matrix) of the total execution time. The QS20 and QS22 blades achieves the highest bandwidth utilization on average across the different values of n owing to their DMA based data transfer mechanism with a large chunk size. The QS20 blade (with Rambus XDR) achieves higher bandwidth utilization than the QS22 blade (with DDR2). The 4P Barcelona system achieves significantly higher sustained bandwidth and bandwidth utilization than the 2P Harpertown system. This exemplifies the scalability benefit of the NUMA architecture. The AMD Barcelona equips optimized memory access scheduling algorithms for interleaved streaming accesses, and this also contributes to higher bandwidth utilization. However, the 2P Harpertown system delivers higher flop rates per unit bandwidth consumption owing to the large shared cache memory.

The blocked implementations yield significantly better results than the initial implementation, as shown in the bottom half of Figure 3, but still deliver significantly lower performance than the theoretical peak. In the case of the x86 architectures, there are a number of possible explanations. First, we need to tune the blocking with respect to the different sizes and associativities at all levels of the cache hierarchy to achieve higher performance. This task would be daunting task even for skilled programmers. Secondly, the blocked implementation may interact, and may even interfere, with the various hardware mechanisms in an unintentionally negative way. For example, the blocked version has a more complex memory access pattern, which may reduce the effectiveness of the hardware prefetchers. Thirdly, the behavior of the memory system mechanisms are complex and challenging to reason about. For instance, on the Harpertown, data in DRAM is first read into the lower level (L2) cache, whereas it is read into the highest level (L1) cache first and moved to the non-inclusive L2 and L3 caches (when the cache line is evicted) on the Barcelona. All these differences can affect the performance in non-intuitive ways and this imposes challenges to a programmer if one wishes to extract the highest achievable flop rates out of the chip.

In contrast, for the Cell/B.E. based systems, it is significantly easier to understand the data transfer related performance issues owing to its simple architecture. Still, to achieve the highest flop rate, programmers need to consider their code at the assembly level. Each iteration of the innermost loop in Kernel1 requires two vector loads, one vector stores, two address increments, and one vector FMA (fused-multiply-and-add) instructions. As the result, the fixed-point instructions and load/store instructions can become a performance bottleneck. Extensive low level tuning is required to balance the even and odd pipelines and minimize the address calculation overhead.

For the Tesla C1060, the delivered performance is lower than 10% of its theoretical peak even in the best case. The off-loading overhead (11%-70% of the total execution time, which increases as n decreases), integer instructions for address increments, and the lack of additional multiply instructions to feed the SFU lower the deliverable performance. Also, to load data to the block array in the shared memory, the Tesla C1060 needs to calculate the address and issue a load instruction for every single real number even though DRAM access latency can be in principle efficiently hidden with the hardware multithreading mechanism; thus the device memory to the shared memory traffic cannot be perfectly overlapped with the computation as is the case of the Cell/B.E. The Tesla C1060 architecture is less transparent than the Cell/B.E.'s, and so the performance impact of the tunable parameters (e.g. thread block size, block width and height in the blocking approach, and loop unrolling factors) are more difficult to predict, thereby requiring explicit search and tuning.

Figure 4 shows the results for double-precision. Interestingly, we can see that the sustained bandwidth for the Tesla C1060 is higher than the single-precision case, largely due to the increased granularity of data transfer from 64 byte (16 threads in a half warp $\times$ 4 byte floating point number) to 128 byte (16 $\times$ 8 byte floating point number). In the case of double-precision, the QS22 blade delivers higher performance than the QS20 blade, and the QS20 blade becomes compute-bound. We can also note that QS20 and the Tesla C1060 achieve a significantly higher fraction of its theoretical peak than the single-precision case. This shows that the QS20 and the Tesla C1060's peak double-precision performances are easier to achieve than their single-precision counterpart.

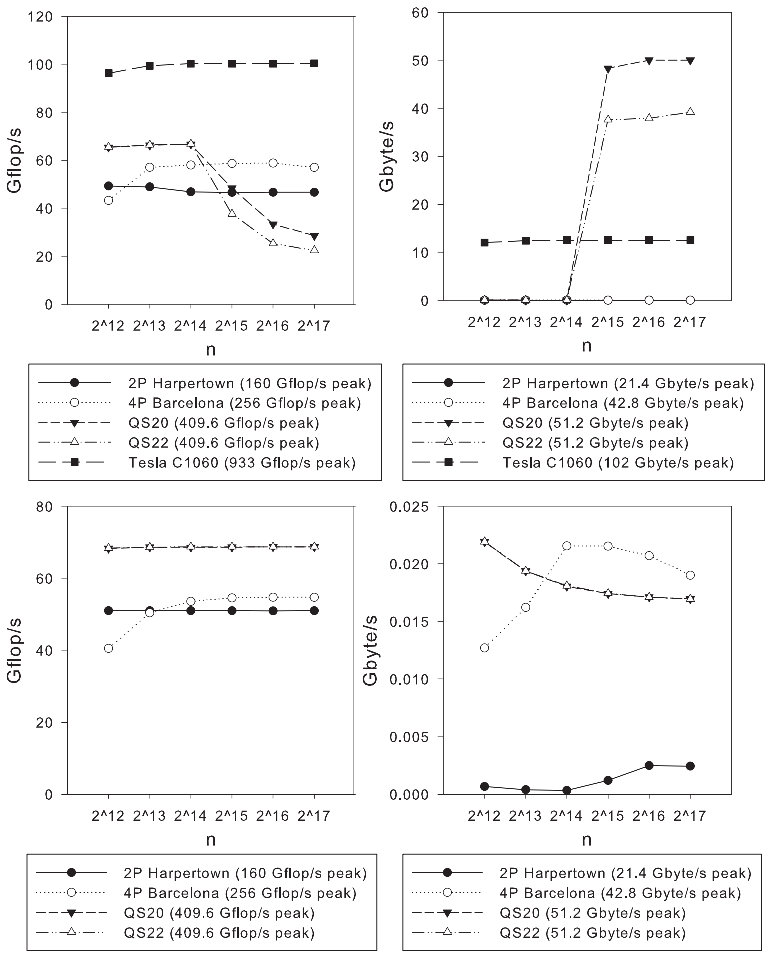


Figure 5: Sustained Gflop/s (left) and bandwidth utilization (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel2 (single-precision). Here, $n_X = n_Y = 128$.

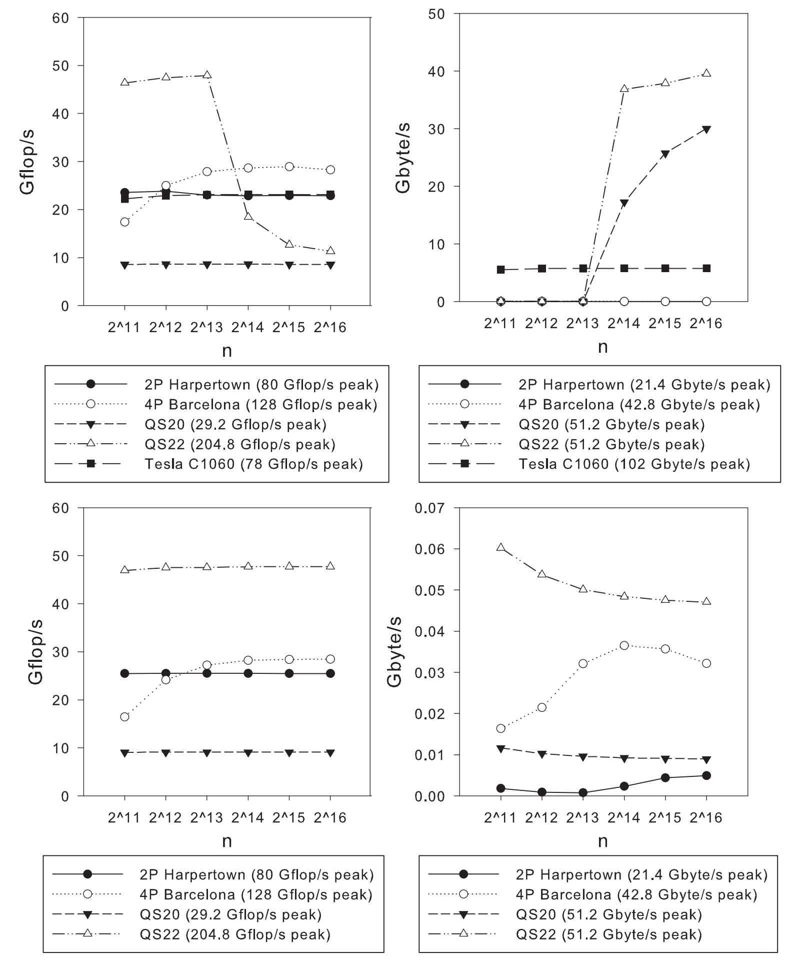


Figure 6: Sustained Gflop/s (left) and bandwidth utilization (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel2 (double-precision). Here, $n_X = n_Y = 128$.

2.4.2 Kernel2

Figures 5 and 6 summarize the results for Kernel2 in single- and double-precisions, respectively. This kernel is highly compute-intensive, and if a pair of two rows can fit into the on-chip memory, Kernel2 becomes compute-bound even without blocking. For single-precision, the Tesla C1060 delivers nearly 100 Gflop/s. Still, this number is significantly lower than the advertised peak numbers, for similar reasons discussed in the case of Kernel1 in addition to *sign()* function. Kernel2's invocation of *sign()* is replaced with predicated instructions; however, computing predication also requires additional cycles. In addition, as the SP has only one execution pipeline and does not have separate integer execution units, performance is more susceptible to the mix with integer instructions than on other architectures. For the QS20 and QS22 blades, the kernel is compute-bound for small n and bandwidth-bound for large n without blocking. In contrast, the QS20 and QS22 blades become compute-bound even for large n with blocking. In the bandwidth-limited cases, the QS20 blade achieves nearly full bandwidth utilization while the QS22 blade delivers approximately 80% of the theoretical peak bandwidth. The QS20 blade uses XDR DRAM and the QS22 blade uses DDR2 DRAM, which explains the reason for the different sustained bandwidth.

For the Tesla C1060, the kernel becomes compute-bound, even without blocking, because of the efficient use of the shared memory. Indeed, blocking does not improve performance, and so we need not consider it further.

2.4.3 Kernel3

Kernel3 is highly floating-point intensive, and its performance is given in Figure 7 and 8. For single-precision, the QS20 blade, the QS22 blade, and the Tesla C1060 card deliver comparable performance for small n. The QS20 and QS22 blade's performance drop sharply with large n without blocking, but the flop rates remain nearly constant with blocking. Considering that our kernel has more adds and subtracts than multiplies, the QS20 and QS22 blades demonstrate near the maximum achievable floating-point performance. Yet, the performance for the Tesla C1060 is only one-quarter of its advertised peak performance. For double-precision, the Tesla C1060 delivers over one-half of its peak flop rates in comparison to one-quarter in the single-precision case as double-precision SP can be solely used for floating-point operations and single-precision SPs can be exploited for integer instructions (*e.g.* address calculation).

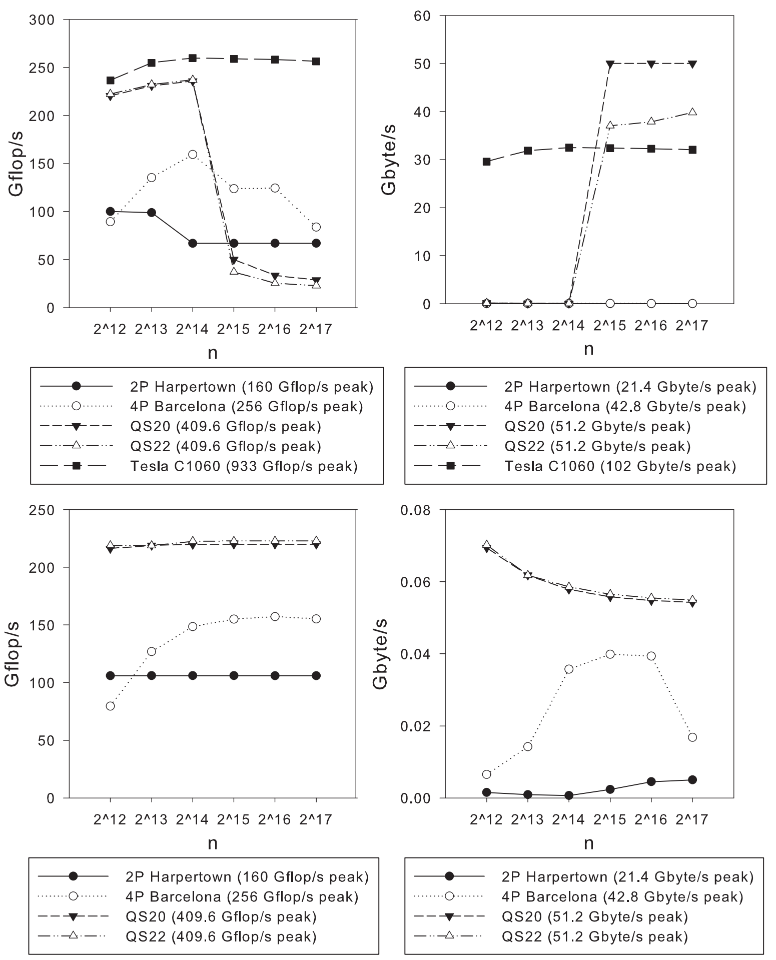


Figure 7: Sustained Gflop/s (left) and bandwidth utilization (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel3 (single-precision). Here, $n_X = n_Y = 128$.

2.5 Summary

In this chapter, we empirically evaluate fundamental design trade-offs among the most recent multicore processors and accelerator technologies. Our primary aim is to aid application designers in better mapping their software to the most suitable architecture, with an additional goal of influencing future computing system design. We specifically examine five architectures, based on: the Intel quad-core Harpertown processor, the AMD quad-core Barcelona processor, the Sony-Toshiba-IBM Cell Broadband Engine processors (both the first-generation chip and the second-generation PowerXCell 8i), and the NVIDIA Tesla C1060 GPU. We illustrate the

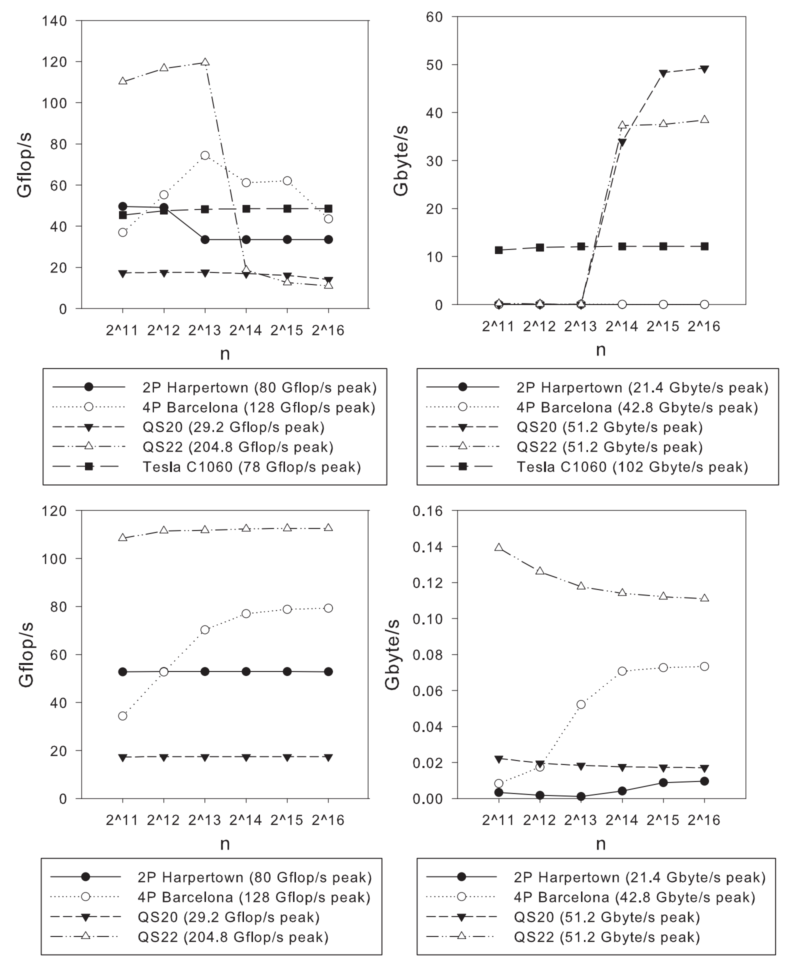


Figure 8: Sustained Gflop/s (left) and bandwidth utilization (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel3 (double-precision). Here, $n_X = n_Y = 128$.

software implementation process on each platform for a set of widely-used kernels from computational statistics that are simple to reason about; measure and analyze the performance of each implementation; and discuss the impact of different architectural design choices on each implementation.

CHAPTER III

AN EFFICIENT TRANSACTIONAL MEMORY ALGORITHM FOR COMPUTING A MINIMUM SPANNING FOREST OF SPARSE GRAPHS

This chapter discusses the match between irregular graph algorithms and Transactional Memory. This chapter is based on the materials in *Seunghwa Kang and David A. Bader, "An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs," The 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Raleigh, NC, February 14th-18th, 2009.*

Data synchronization is one of the most prominent bottlenecks for the widespread and efficient use of multicore computing. A major class of supercomputing applications is based on array data structures and have regular data access patterns; thus data partitioning for parallel programming is relatively simple, and data synchronization is required only at the boundaries of partitioned data. In contrast, another class of important applications uses pointer-based irregular data structures. This raises complex data synchronization issues and significantly exacerbates the situation.

Researchers have attempted to solve this problem in several directions targeting a spectrum of programming efforts and flexibility trade-offs. Thread Level Speculation [130, 123] extracts an additional level of parallelism without programmer intervention by speculatively executing instructions with unresolved data dependency. Kulkarni et al. [71] have developed the Galois run-time system to automatically parallelize irregular applications with user provided hints. However, to achieve the highest level of flexibility or chip utilization over widely varying applications using ever increasing number of cores, explicit parallel programming is often mandated.

Lock and barrier are the most popular data synchronization primitives for parallel programming but are problematic in programming complexity, scalability, or composability for complex applications. Coarse-grained locking has affordable programming complexity but often sequentializes program execution. Fine-grained locking provides superior scalability at the expense of notoriously complex programming. Frequent invocation of barrier primitives can be a significant performance bottleneck as well. Also, for lock-based programs, even correct implementations cannot be simply combined to form larger programs [51].

Transactional Memory (TM) [55] attempts to remedy this problem and promises to provide the scalability of fine-grained locking at the programming complexity of a single lock approach. TM also has great potential to resolve the composability problem [51]. Within a TM framework, we can seamlessly combine different transaction-based codes while maintaining the atomicity of the transactions. Moreover, the assumption under TM fits well to the requirements to benefit from parallelization. von Praun et al. [125] classified applications based on data dependence densities and argued that applications with high data dependency cannot benefit significantly from parallelization while applications with low data dependency can. TM also becomes more efficient as the degree of data dependency drops, and this suggests that if an application can benefit from parallelization, then that application can also benefit from TM.

Programmers only need to demarcate the critical sections for shared data accesses based on TM semantics. Then, if there is no conflict in the shared data accesses, the underlying TM system executes critical sections (or transactions) in parallel. If there are conflicts, the affected transactions are aborted and re-executed to provide the semantics of sequential execution for transactions. TM systems based on software (STM) [54, 78, 43], hardware (HTM) [50, 10, 104, 86, 22], and hybrid (HyTM) [39, 84, 116] approaches have been proposed, which have trade-offs in flexibility, dedicated hardware requirements, and performance overhead for different stages of deployment. There are already multiple open source STM implementations. Sun Microsystems had been expected to release its Rock processor [121], the first microprocessor with hardware support for accelerating TM, though the project was canceled during the Oracle's acquisition process.

Up to this point, most research on Transactional Memory has focused on systems and architectures, but few have considered its impact on algorithms and applications. There are several existing benchmarks [36, 49, 41, 101] for TM systems, but these applications do not reveal the full potential of TM for real-world applications. Scott et al. [115] implemented Delaunay triangulation using their TM system as a real-world application. Yet, for their implementation, data synchronization is required only at the boundaries of partitioned data. Watson et al. [126] reported an implementation of Lee's circuit routing algorithm using TM. They achieved a high level of parallelism using TM in combination with deep understanding of the algorithm to reduce the number of aborted transactions.

In this chapter, we investigate the impact of Transactional Memory on algorithm design for applications with intensive data synchronization requirements. We believe that TM has great potential to facilitate scalable parallel algorithm design without involving the complexity of fine-grained locking or the ingenuity of lockfree algorithms. We provide an algorithmic model for TM to assist the development of efficient TM algorithms. Perfumo et al. [101] suggested *commit phase overhead*, *read-to-write ratio*, *wasted time*, *useful work*, and *cache accesses per transactional access* as metrics for characterizing STM applications. System researchers may feel comfortable with these metrics, but algorithm designers may not. Our model aims to reveal the key features of TM systems to be considered in designing algorithms. Then, we move our focus to graph algorithms. Graph algorithms are typically pointer-based and have irregular data access patterns with intensive data synchronization requirements. Graph algorithms also have interesting characteristics that fit well to TM systems at the same time. As a case study, we present an efficient Transactional Memory algorithm for computing a minimum spanning forest (MSF) of sparse graphs. Our algorithm adopts a key idea from Bader and Cong's MSF algorithm [13], which implements a lockfree parallel MSF algorithm by combining Prim's algorithm with Borůvka's algorithm. Our algorithm replaces the Borůvka's algorithm part with tree merging and exploits TM to achieve high scalability with low performance overhead. We present our algorithm with special emphasis on TM specific design issues. We implement and test our algorithm using the Stamp [83] framework based on Transactional Locking II [43] STM. Our implementation demonstrates remarkable speedup for different sparse graphs throughout the experiments. Yet, as our algorithm executes a significant fraction of the code inside transactions, the high STM overhead nullifies the speedup. This reveals the limitation of STM and highlights the necessity of the low overhead hardware support. Dice *et al.* [42] had experimented our algorithm on the Rock processor—with few modifications to work around the Rock processor's limitations—and achieved both high scalability and high-performance. The result confirms our argument.

3.1 Transactional Memory and Algorithm Design

3.1.1 Transactional Memory and Lockfree Algorithms

Lock is a popular synchronization primitive in multi-threaded programming. If multiple threads access shared variables concurrently, fine-grained locking is required to avoid sequentialization on a single lock. However, this approach involves a notoriously difficult programming challenge. Remembering the relationship between multiple locks and multiple shared variables is already a non-trivial issue. Deadlock and livelock problems, starvation, and other related issues, often increase the programming complexity to a prohibitive level. Therefore, many algorithm designers use coarse-grained locking instead and make an effort to reduce the size of critical sections to minimize the sequentialization problem. Unfortunately, this does not always work especially with a large number of cores. Ambitious researchers devote their effort to write lockfree algorithms that avoid race conditions using novel algorithmic innovations instead of locks. Yet, lockfree algorithm design requires deep understanding of the algorithm and the underlying system. As a consequence, lockfree algorithms are usually hard to understand and often rely on architecture-specific assumptions. Also, lockfree algorithms often involve additional computation to avoid data races.

Transactional Memory has great potential to change this situation. Programmers only need to mark the start and the end of transactions in their explicitly parallel code. Then, the TM system will automatically run non-conflicting transactions in parallel to achieve high-performance while roll-back and re-execute conflicting transactions to provide the semantics of sequential execution for transactions. If the TM system can support this mechanism with high scalability and low performance overhead, we may not need to rely on lockfree algorithms anymore. Bader and Cong's parallel MSF algorithm [13] is a lockfree algorithm and incurs the overheads discussed above. As an illustration of our arguments, we design a new algorithm for computing a MSF using Transactional Memory. Section 3.3 further discusses the topic.

3.1.2 Algorithmic Model for Transactional Memory System

Transactional Memory can help programmers to write explicitly parallel programs, but TM cannot guarantee scalable performance for every implementation. To best exploit TM, we need to write an algorithm that fits well with the underlying TM framework. However, it is difficult for algorithm designers to decide which design

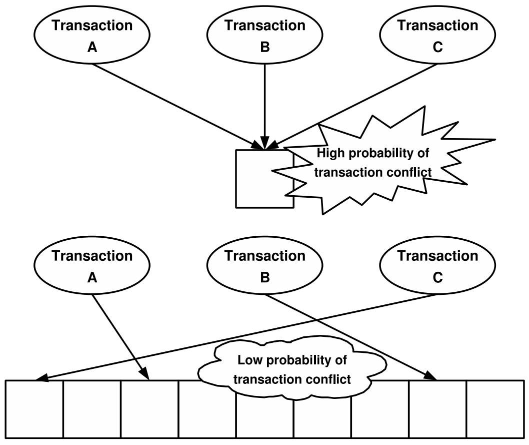

Transaction A
Transaction B
Transaction C
High probability of transaction conflict
Transaction A
Transaction B
Transaction C
Low probability of transaction conflict

Figure 9: When multiple transactions access one shared variable at the same time, those transactions are highly likely to conflict (top). In contrast, if concurrent transactions mostly access different shared variables, those transactions will conflict with low probability (bottom).

choice will lead to the best result without in depth knowledge of TM systems. We design our algorithmic model for TM towards this goal. Our model focuses on revealing the common key features of widely varying TM systems, which need to be considered in algorithm design. Our model does not aim to estimate precise execution time as this complicates the model without providing significant intuition in algorithm design.

Let, $T_{par.}$ and $T_{seq.}$ denote parallel and sequential execution time, respectively, and p is the number of concurrent threads (assuming w.l.o.g. one thread per core). Then, the following equation states our model assuming that an algorithm does not have an inherently sequential part.

$$\begin{aligned}
T_{par.} &= \frac{T_{tr.} + T_{non-tr.}}{p} \\
T_{tr.} &= T_{seq.} \times \sum_{i=1}^{\#\ transactions} (\tau_i \times \#\ trials \times ov(s_i)) \\
T_{non-tr.} &= T_{seq.} \times (1 - \tau) \\
\#\ trials &= \sum_{j=1}^{\infty} j \times {p_i}^{j-1} \times (1 - p_i) \\
\tau &= \sum_{i=1}^{\#\ transactions} \tau_i
\end{aligned}$$

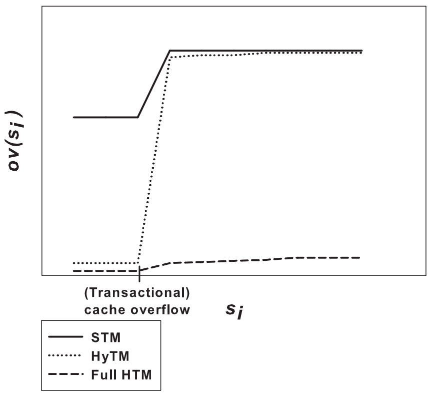


Figure 10: Transactional overhead $ov(s_i)$ varies as a function of transaction size s_i. This figure assumes per core transactional cache for HyTM or full HTM and two-level memory hierarchy.

where, $\frac{T_{tr.}}{p}$ and $\frac{T_{non-tr.}}{p}$ denote the execution times for the transactional part and the non-transactional part of an algorithm with p threads. An algorithm may consists of multiple transactions ($\#\ transactions$), which account for different fractions of the total number of operations to run the algorithm (τ_i) and conflict with other transactions with the probability p_i. Transactions may execute several times to commit ($\#\ trials$) due to conflicts. p_i in each trial may vary according to the system specific contention management schemes. Also, an operation inside a transaction runs $ov(s_i)$ times slower than an operation outside a transaction owing to the TM overhead, where s_i is the transaction size. We rely on algorithm designers to estimate the value of p_i, as algorithm designers are likely to best understand the data access patterns of their own algorithm. To systematically estimate the level of data conflict in the legacy code, we refer the readers to von Praun et al.'s work [125]. $ov(s_i)$ is specific to the underlying TM system and also varies as the function of the transaction size. Figure 10 portrays the overhead assuming per core transactional cache. If the underlying TM system has a shared transactional cache, the overhead becomes a function of the aggregate concurrent transaction size.

Our model provides several guidelines to algorithm designers. STM incurs large $ov(s_i)$ regardless of the transaction size, and this suggests that algorithm designers should focus on minimizing τ. Yet, this increases the programming complexity and hampers the benefit of Transactional Memory. Moreover, if τ is small, a single lock approach may suffice to achieve scalability. The key disadvantage of a single lock approach is the sequentialization of critical sections on a single lock. This may not significantly limit the scalability of the algorithm if every thread spends only a negligible fraction of the time inside critical sections. HyTM incurs only low overhead for transactions that fit to the transactional cache. The excessively large transactions still can incur high overhead. Full HTM may able to provide relatively low overhead even in the case of overflow, and Ananian et al. [10] presented UTM (Unbounded Transactional Memory) towards this goal. However, full HTM requires significant modification to the existing microprocessor architecture or even the DRAM architecture. This may not happen within the foreseeable future. Even with the full HTM support, an excessively large transaction can still be a problem as it spans a longer period of time and accesses multiple memory locations; thus likely to get aborted multiple times or aborts many other transactions and increases p_i. In summary, our model suggests that algorithm designers should focus on minimizing p_i while avoiding excessively large transactions.

3.1.3 Transactional Memory and Graph Algorithms

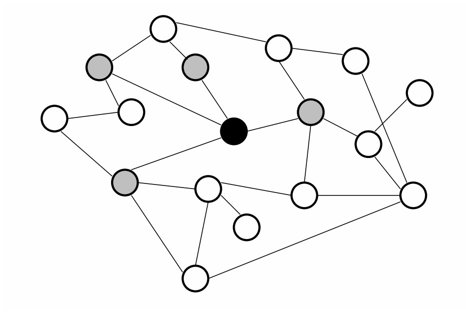


Figure 11: A common graph operation: **pick one vertex** (black) and **find (and apply operations on) its neighbors** (gray)

Graph algorithms represent one important class of applications that use pointer-based irregular data structures and have multiple features that can take advantage of Transactional Memory. While some of the synthetic graphs, scientific meshes, and regular grids can be easily partitioned in advance [71], many other graphs representing real-world data are hard to partition and thus, are challenging to efficiently execute on parallel systems [18]. Many graph algorithms also finish their computation by visiting multiple vertices in the input graph, and even with the optimal partitioning, we can traverse from one partition to the other partitions with a relatively small number of hops. Therefore, any vertex in the graph can be concurrently accessed by multiple threads, and this necessitates data synchronization over the entire graph. However, if the graph size is large enough relative to the number of threads, two or more threads will access the same vertex at the same time with very low probability. This leads to low p_i in our TM model. Then, TM can abstract the complex data synchronization requirements involving a large number of vertices, and high scalability can be achieved owing to the low degree of data contention.

In addition, we can easily decompose many graph algorithms to multiple transactions. Figure 11 depicts a common graph operation: **pick one vertex** and **find (and apply operations on) its neighbors**. Many graph algorithms iterate this process, and this common routine can be implemented as a single transaction. Thus, an algorithm can be implemented as iterations of transactions. This leads to large τ and exemplifies the key benefit of TM over lock-based approaches. A single lock approach will sequentialize nearly the entire computation while fine-grained locking involves prohibitively difficult programming challenges. In contrast, TM can significantly reduce the gap between high level algorithm design and its efficient implementation, as the independent tasks can easily map to transactions without compromising scalability. However, if the underlying TM system incurs high overhead, this overhead can nullify the high scalability as most operations need to be executed inside transactions. This necessitates a low overhead hardware mechanism to obtain superior performance from high scalability. We may see low overhead HTMs (or HyTMs) within the near future, but excessively large transactions are likely to incur high overhead for longer time period, if not forever, as full HTM design is more challenging and unlikely to be commercialized in the foreseeable future. If we decompose graph algorithms in the above way, a vertex degree determines the size of a transaction, and for sparse graphs, most transactions have relatively small size; thus, transactional cache overflow may not be an issue. Still, there can be a small number of vertices with a exceptionally large vertex degree, and this can be a performance bottleneck. In our MSF algorithm, we modify the algorithm to address this issue, and this is further discussed in Section 3.3.2.

3.2 Minimum Spanning Forest Algorithm for Sparse Graphs

3.2.1 Sequential Minimum Spanning Forest Algorithm

Given a fully connected undirected graph, minimum spanning tree (MST) algorithms find a set of edges which fully connects all vertices in the graph without cycles while minimizing the sum of edge weights. If the input graph has multiple connected components, minimum spanning forest (MSF) algorithms find multiple MSTs, one

MST per each connected component. MST has multiple practical applications in the area of VLSI layout, wireless communication, distributed networks, problems in biology and medicine, national security and bioterrorism, and is often a key module for solving more complex graph algorithms [13]. Prim's, Kruskal's, and Borůvka's algorithms are three well known approaches for solving MST problems. Even though there are other algorithms with lower asymptotic complexity, they run slower on real systems than these algorithms [88] due to large hidden constants in the asymptotic complexities. Based on the experimental results in [88, 13], there is often no clear winner among these three algorithms as execution time depends on the topology of the input graph and the characteristics of the computing system.

3.2.2 Parallel Minimum Spanning Forest Algorithm

Prim's and Kruskal's algorithms are inherently sequential, while Borůvka's algorithm has natural parallelism. Therefore, most previous parallel MST algorithms are based on Borůvka's approach (see Bader and Cong's paper [13] for a summary). Even though these algorithms' achieve parallel speedup for relatively regular graphs, none of these Borůvka's algorithm-based implementations runs significantly faster than the best sequential algorithm for a wide range of irregular and sparse graphs. Bader and Cong [13] also introduced a new parallel MST algorithm which marries Prim's algorithm with Borůvka's algorithm. This new algorithm grows multiple MSTs using Prim's algorithm until conflict occurs. While growing MSTs, each thread marks the vertices in its own MST as visited and also colors all neighbors of the marked vertices with its own color. If the algorithm encounters a vertex marked or colored by other threads, conflict occurs. Then, the thread starts to grow a new MST with a new vertex. If there is no remaining unmarked and uncolored vertex, the algorithm switches to the Borůvka's algorithm stage, finds the connected components, and compacts the graph. The algorithm iterates this process until the number of remaining vertices falls below the given threshold value. When this happens, the algorithm runs the best sequential algorithm to finish the computation. If the input graph has multiple connected components, this algorithm finds an MSF of the graph. Actually, if there is only one thread, this algorithm behaves similar to Prim's algorithm while it works as Borůvka's algorithm if the number of threads is equal to the number of vertices.

This algorithm avoids data races without using locks. To achieve this goal, this algorithm uses two arrays, *visited* and *color*. However, it is not trivial to fully understand the mechanisms to avoid data races. Also, this algorithm assumes sequential consistency for the underlying system, and if the underlying system supports only the relaxed consistency model, additional fence operations are required. Therefore, even assuming that a high level description of the algorithm is given, expertise in both algorithm and architecture is required to implement the algorithm correctly.

Moreover, the lockfree nature of the algorithm is achieved at the cost of additional performance overhead. If a vertex is colored by other threads but not marked as visited, adding this vertex to the thread's own MST may not lead to true conflict in most cases. However, this algorithm takes a conservative position and treats this case as a conflict to avoid possible race conditions. If the input graph has a relatively small diameter, this can lead to an excessive number of conflicts, and the algorithm may not perform much useful work in the Prim's algorithm stage. Then, the Prim's algorithm stage just wastes computing cycles, and the Borůvka's algorithm stage performs the most useful work. Even when the graph has a large diameter, and conflicts occur infrequently, this algorithm runs both Prim's algorithm and Borůvka's algorithm and also additional computation to achieve its lockfree nature; this leads to additional performance overhead. Due to this overhead, this algorithm requires 2 to 6 processors to outperform the best sequential algorithm depending on the input graph.

3.3 An Efficient Transactional Memory Algorithm for Computing a Minimum Spanning Forest of Sparse Graphs

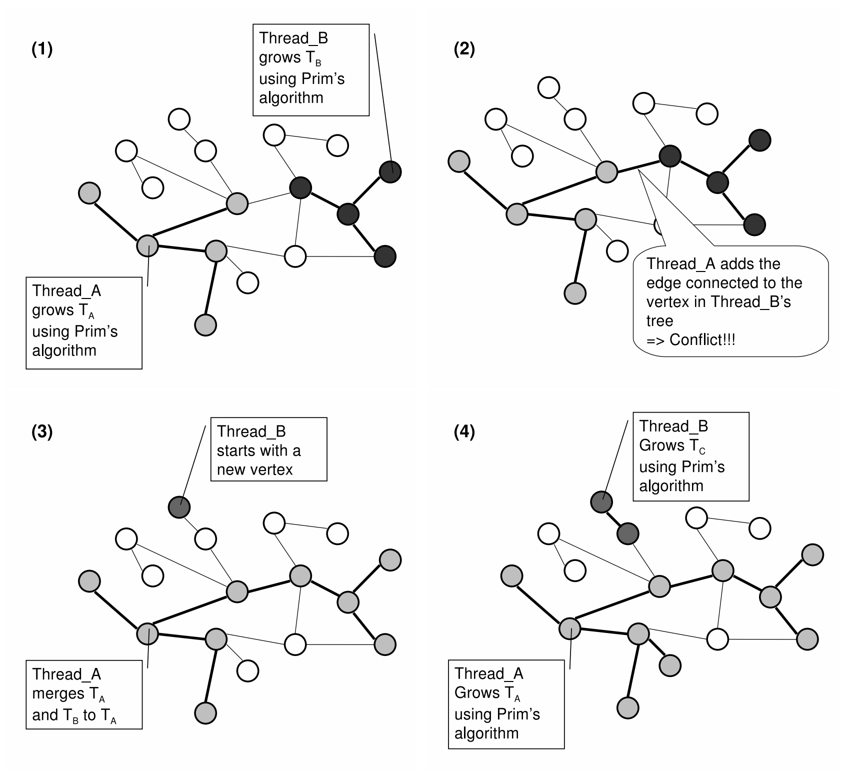

(1)
Thread_B grows T_B using Prim's algorithm
Thread_A grows T_A using Prim's algorithm
(2)
Thread_A adds the edge connected to the vertex in Thread_B's tree => Conflict!!!
(3)
Thread_B starts with a new vertex
Thread_A merges T_A and T_B to T_A
(4)
Thread_B Grows T_C using Prim's algorithm
Thread_A Grows T_A using Prim's algorithm

Figure 12: A high level illustration of our MSF algorithm

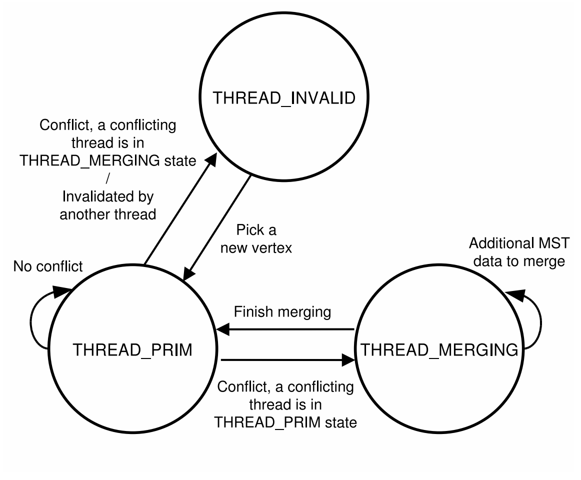

THREAD_INVALID
THREAD_PRIM
THREAD_MERGING
Conflict, a conflicting thread is in THREAD_MERGING state / Invalidated by another thread
Pick a new vertex
No conflict
Finish merging
Conflict, a conflicting thread is in THREAD_PRIM state
Additional MST data to merge

Figure 13: A State transition diagram for our MSF algorithm

In our new MSF algorithm, each thread runs Prim's algorithm independently similar to Bader and Cong's algorithm [13]. However, our MSF algorithm does not

```
while true do
    TM_BEGIN();
    if current thread's state is THREAD_PRIM then
        $w = heap\_extract\_min(H)$; /* pick one vertex */
        if w is already in the current thread's MST then
            TM_END();
            continue();
        end
        if w is unmarked by other threads then
            Add w to the current thread's MST and insert its neighbors to the heap
            H. /* find (and apply operations on) its neighbors */
            TM_END();
        else
            /* w is already marked by another thread */
            if the other thread marked w is in THREAD_PRIM state then
                Steal the other thread's MST data and invalidate that thread.
                TM_END();
                break; /* go to THREAD_MERGING state */
            else
                /* the other thread marked w is in THREAD_MERGING state */
                Release own MST data to that thread.
                TM_END();
                break; /* go to THREAD_INVALID state */
            end
        end
    else
        /* invalidated by other thread */
        TM_END();
        break; /* go to THREAD_INVALID state */
    end
end
```

Algorithm 1: Pseudo-code for THREAD_PRIM state. TM_BEGIN() marks the start of a transaction, and TM_END() marks its end.

have the Borůvka's algorithm stage. Instead, if one thread attempts to add a vertex in another thread's MST, or conflict occurs, one thread merges two MSTs, and the other thread starts Prim's algorithm again with a new randomly picked vertex. Our new algorithm also does not detect conflicts in a conservative way as Bader and Cong's algorithm. Instead, our new algorithm relies on the semantics of transactions to avoid data races. Figure 12 illustrates our algorithm in high level, and Figure 13 and Algorithm 1 give further details. There are three states: THREAD_INVALID, THREAD_PRIM, and THREAD_MERGING. A thread starts in THREAD_INVALID state, and changes its state from THREAD_INVALID state to THREAD_PRIM by assigning a new color for the thread and also randomly picking a new start vertex. Then it runs Prim's algorithm in THREAD_PRIM state, and grows its MST (newly added vertices are colored with the thread's color). In this state, we iterate a common graph operation, **pick one vertex** and **find (and apply operations on) its neighbors**, as a single transaction as depicted in Algorithm 1. When two threads (assume $Thread\_A$ and $Thread\_B$) conflict, we merge the two trees of the two conflicting threads by merging their MST data. When $Thread\_A$ detects a conflict and finds $Thread\_B$ is in THREAD_PRIM state, $Thread\_A$ invalidates $Thread\_B$ and merges the MST data of the two threads. $Thread\_B$ restarts by picking a new vertex. When $Thread\_A$ detects a conflict and finds $Thread\_B$ is in THREAD_MERGING state, $Thread\_A$ releases its MST data to $Thread\_B$ and moves to THREAD_INVALID state to restart with a new vertex. In a single-threaded case, our algorithm runs nearly identical to sequential Prim's algorithm with small additional overhead related to the state management.

To implement our algorithm, a complex data synchronization issue arises. Every vertex can be accessed concurrently, and every thread's state variable can be modified by multiple threads at the same time. To implement this algorithm with fine-grained locking, one thread may need to acquire up to four locks (one for its own state variable, two for the source and destination vertices in a new MST edge, and one more for the conflicting thread's state variable). This can lead to many complex scenarios that can cause race conditions, deadlocks, or other complications, and it is far from trivial to write correct and scalable code. In contrast, Transactional Memory can gracefully abstract all the complications related to the data synchronization issues in our algorithm. As our algorithm executes a large fraction of the code inside transactions (τ is large), this may not fit well with STM, but future TM systems with efficient hardware support will resolve this problem. Also, if the input graph is large enough, the level of data contention is low (low p_i in our TM model), and our algorithm fits well with TM systems.

However, our algorithm has other sources for parallel overhead. First, we need to pick a new start vertex multiple times. If a newly picked vertex is already included in other threads' MST, we need to pick a new one again. Second, MST data merging can take a significant amount of time. The first overhead may not be significant at the beginning of the algorithm, and even at the end of the algorithm (when almost every vertex is included in other threads' MST), this does not significantly slow down other threads' execution if we ignore the impact on the memory subsystem. To minimize the impact on the memory subsystem, if a thread picks a vertex in another thread's MST, we suspend the thread for a short time before picking a new vertex. Therefore, we need to focus on estimating and minimizing the overhead of MST data merging.

3.3.1 MST Data Merging and Composability

Assuming *Thread_A* merges its own MST data with *Thread_B*'s MST data, MST data merging consists of two tasks. First, as every vertex in *Thread_A* and *Thread_B*'s MST needs to be marked with a same color after the merging, *Thread_A* needs to recolor all the vertices in *Thread_B*'s MST to *Thread_A*'s color. Second, *Thread_A* needs to merge its own heap (a heap data structure is maintained to find the minimum weight edge efficiently as other Prim's algorithm based implementations) with *Thread_B*'s heap. We can also expect that there will be a few merges of large MST data at the beginning of the algorithm followed by more frequent merges of large MST data with small MST data at the end of the algorithm.

If we recolor all the vertices in other thread's MST in a naïve way, it will significantly increase the parallel overhead of our algorithm. Instead, we add one level of indirection. We create a color array that maps an index to a color. When we add a new vertex to the MST, we mark that vertex with the index to the color array element for the thread instead of the color of the thread. Thus, for re-coloring, we need to only update the color array elements for the MST data of $Thread\_B$. The number of color array elements to be updated is identical to the number of MST data mergings that happened in $Thread\_B$'s MST data.

For actual implementation, this requires two additional shared data arrays to map an index to a color and a color to the owner thread. Under fine-grained locking, this requires additional lock arrays, and we need to re-design the entire lock acquisition protocol to avoid dead-lock or other lock related issues. Within a TM framework, in contrast, we can easily extend our algorithm without re-designing data synchronization schemes. This exemplifies TM's benefit over lock in composability.

To exploit the fact that there will be more frequent merges of large and small MST data, we switch $Thread\_A$'s MST data with $Thread\_B$'s MST data before merging if $Thread\_B$'s heap is larger than $Thread\_A$'s heap. Then, we merge two heaps by inserting the elements of the smaller heap ($Thread\_B$'s heap) to the larger heap ($Thread\_A$'s heap). These inserts are the most expensive parallel overhead in our algorithm.

We grow multiple MSTs concurrently, and this involves the overhead of MST data merging. However, this also decreases the average heap size during the execution. Instead of one thread growing a large MST to the end, there will be multiple small MSTs grown by multiple threads. A small MST has less elements in its heap. If a heap has n elements, heap inserts or extracts costs $O(log(n))$ memory accesses. Therefore, a single heap operation costs less for a smaller MST. Especially, if a heap does not fit into the cache memory, multiple non-contiguous memory accesses in a single heap operation leads to multiple cache misses, and these cache misses can be the most expensive cost of the algorithm. Combined with the modern cache subsystem with multiple layers, the smaller heap size can have more significant impact on the performance than it appears in the asymptotic notation. If the impact of this is larger than the MST data merging overhead, our algorithm can scale super-linearly.

3.3.2 Avoiding Excessively Large Transactions and Strong Atomicity

As discussed in Section 3.1.2, excessively large transactions are undesirable. There are two sources for large transactions in our algorithm. First, if we merge tree data inside a transaction, this creates a very large transaction. Second, if there are high degree vertices, this leads to large transactions.

In our state transition diagram, if a thread in THREAD_PRIM state (say $Thread\_A$) attempts to add a vertex in the MST of a thread in THREAD_MERGING state (say $Thread\_B$), $Thread\_A$ invalidates itself and appends its own MST data to $Thread\_B$'s queue for MST data. If MST data are fetched out from the queue for merging, this data cannot be accessed by other threads, and only the queue access needs to be executed inside a transaction. Therefore, MST data merging in our algorithm does not create a large transaction.

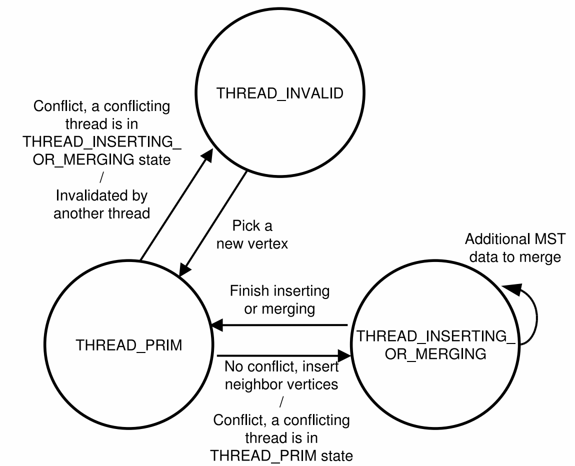

THREAD_INVALID
Conflict, a conflicting thread is in THREAD_INSERTING_ OR_MERGING state / Invalidated by another thread
Pick a new vertex
Additional MST data to merge
Finish inserting or merging
THREAD_PRIM
THREAD_INSERTING_ OR_MERGING
No conflict, insert neighbor vertices / Conflict, a conflicting thread is in THREAD_PRIM state

Figure 14: A Modified state transition diagram for our algorithm

High degree vertices also create a large transaction. If we extract a vertex and insert all neighbors of the extracted vertex in a single transaction, the size of a transaction grows proportional to the vertex degree. This can be solved in a similar way to the first case. If we change our state transition diagram to Figure 14 and insert neighbors outside a transaction, we can avoid large transactions even with high degree vertices.

These changes require accessing shared data both inside and outside transactions. At first glance, as MST data fetched out from the queue can only be accessed by a single thread, one can easily assume that this may not cause a problem. This is true assuming strong atomicity [23] but can be problematic under weak atomicity, which does not define the correct semantics among interleaved transactional and non-transactional code. If a transaction doomed to be aborted reads data updated outside a transaction, which in turn can raise a segmentation fault, or non-transactional code reads data updated by a doomed transaction before it is restored, this can introduce a bug that is hard to find. One can easily assume that accessing shared data outside a transaction is a naïve program bug, but this may not be true if we consider algorithm optimization. Our experience advocates strong atomicity in TM semantics.

3.3.3 Color Filtering

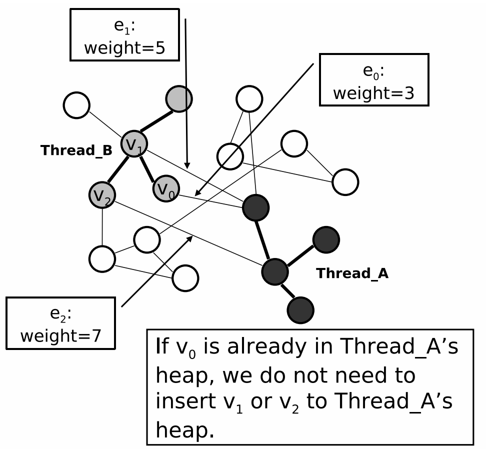

e_1: weight=5
e_0: weight=3
Thread_B
v_1
v_2
v_0
Thread_A
e_2: weight=7
If v_0 is already in Thread_A's heap, we do not need to insert v_1 or v_2 to Thread_A's heap.

Figure 15: Color filtering: filter out unnecessary inserts

Our algorithm works well if the input graph has a large diameter as conflict occurs only infrequently in that case. However, if the input graph has a small diameter and conflict occurs more frequently, the increased MST data merging cost can reduce the performance benefit of parallelization. However, for graphs with a small diameter, there is another opportunity for the optimization. Assume we insert a neighbor vertex into the heap. If the heap already includes a vertex, which connects the current thread's MST to other threads' MST with a lower weight than the new vertex to insert, we do not need to insert the new vertex as it cannot be an MST edge. Figure 15 illustrates the case. We can use the color of a vertex to filter out heap inserts. We maintain an additional data structure that maps a color to the minimum edge weight to connect the current thread's MST to the MST of that color. Considering that the heap operations account for the large fraction of the total execution time, reduced heap inserts can mitigate the increased parallel overhead owing to the frequent conflicts. Actually, this has similar impact to the connected component and compact graph steps in Borůvka's algorithm without incurring the high overhead of those steps.

3.3.4 Heap Pruning

When we merge the heap of two threads (assume $Thread\_A$ and $Thread\_B$), $Thread\_A$'s heap can include vertices in $Thread\_B$'s MST, and $Thread\_B$'s heap can include vertices in $Thread\_A$'s MST. If $Thread\_B$'s heap has less elements, we merge two heaps by inserting $Thread\_B$'s heap elements to $Thread\_A$'s heap. In this case, we do not insert $Thread\_B$'s heap element which belongs to $Thread\_A$'s MST. Yet, $Thread\_A$'s heap can still include vertices in $Thread\_B$'s MST. Thus, after merging, the heap can include vertices in its own MST, which cannot be an MST edge. When we extract a heap element, we check whether the extracted vertex is in its own MST or not, and this does not affect the correctness. Still, if the merging iterates multiple times, and near the end of the algorithm, there can be a very large heap with almost every vertex included in its own MST. At this time, there will be only one thread that includes almost all the vertices in the graph with few remainings. This thread will spend most time for extracting vertices in its own MST, while all the other threads are idling. To prevent this situation, we count the number of extracted heap elements that belong to its own MST. If this number grows above the given threshold, we scan the heap and remove the vertices in its own MST. By combining this heap pruning with the above color filtering, this can significantly reduce the heap size, especially for graphs with a small diameter.

3.4 Experimental Results on STM

3.4.1 Test Graphs

Table 4: Test graphs for experiments. For graph generators that generate edges with uniform weight, we modify the code to generate random weight edges.

graph type	generator	comments
2-D grid	LEDA	-
3-D grid	LEDA	3-D grid has a relatively small diameter.
Web graph	EM-BFS	Web graph with 2K levels, and each level has 3M/2K vertices.
USA West graph	-	USA West roadmap graph. Included as a real word graph.
Random	GTgraph	Edges added by randomly picking two vertices from the entire vertices of the graph. Reveals the worst case behavior of our algorithm except for few pathological cases (Figure 16)

Table 5: A Comparison of the single-threaded execution time (in seconds) with the STM overhead (compiled with *Makefile.stm* in the Stamp framework), single-threaded execution time without STM overhead (compiled with *Makefile.seq* in the Stamp framework), and total MST data merging time in 64 threads case.

	2-D grid	3-D grid	web graph	USA West	Random
with STM overhead	983.1	1197	911.2	1143	1017
without STM overhead	12.96	18.60	12.00	17.33	97.34
MST data merging time	1.386	5.459	1.330	0.2734	59.31

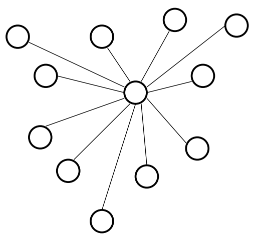


Figure 16: A pathological case for our algorithm.

We use a variety of graphs to experimentally test the performance of our algorithm. These inputs are chosen because they represent diverse collection of real-world instances, or because they have been used in the previous MST studies. We use the road map graph from 9th DIMACS implementation challenge website (`http://www.dis.uniroma1.it/~challenge9`) and also use EM-BFS and GTgraph graph generators available from the DIMACS website in addition to LEDA [81]. The test graphs summarized in Table 4 cover different graphs with varying diameters as well.

3.4.2 Experimental Setup

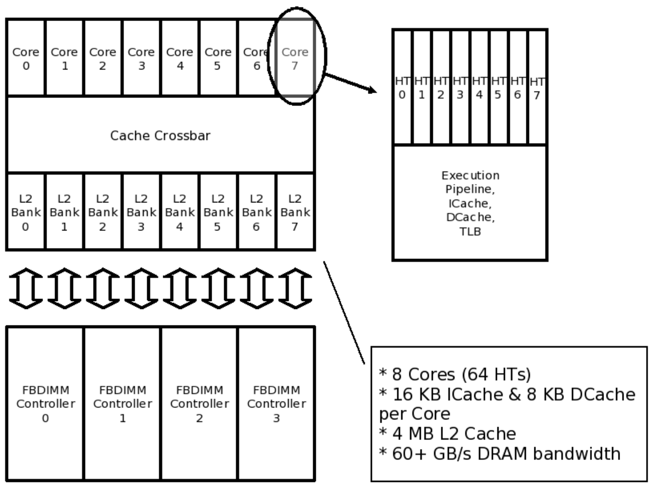


Figure 17: Sun UltraSparc T2 (Niagara 2) processor.

We test our implementation on a system with a Sun UltraSparc T2 Niagara 2 processor (Figure 17) and 32 GB main memory. The Niagara 2 processor has 8 cores, and each core has 8 hardware threads (HTs). 8 HTs share a single execution pipeline. If an application kernel has frequent non-contiguous memory accesses and the execution time is limited by memory access latency, HTs can efficiently share the single execution pipeline without significant performance degradation in single-threaded performance. As our MSF algorithm has multiple non-contiguous memory accesses and the memory latency of the accesses is the key cost of the kernel, the Sun Niagara 2 processor is a suitable architecture for the algorithm.

We use the Stamp (STAMP 0.9.9) framework [83] based on Transactional Locking II (TL2-x86 0.9.5) STM [43] for the implementation and the experiments. We use *gcc 4.0.4* compiler with *-O3 -mcpu=niagara2 -mtune=niagara* optimization flags. We compile the code with *Makefile.stm* (in the Stamp framework) for the STM-based executable and *Makefile.seq* (also included in the Stamp framework) for the executable without STM overhead. Also, we increase the STM lock array size (*_TABSZ* in *tl2.c*) from 2^{20} (default value) to 2^{25} to reduce the number of false transaction conflicts.

3.4.3 Experimental Results

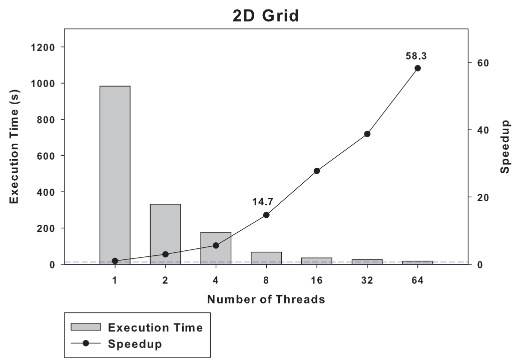


Figure 18: Execution time and speedup for the 2-D grid graph (3.24M vertices, 6.48M edges).

Figures 18, 19, 20, 21, and 22 summarize the execution time and speedup in the experiments. The dashed horizontal line in the figures denotes the single-threaded execution time for running the same algorithm (nearly identical to the sequential Prim's algorithm) without STM overhead. Our algorithm scaled more than 8 times

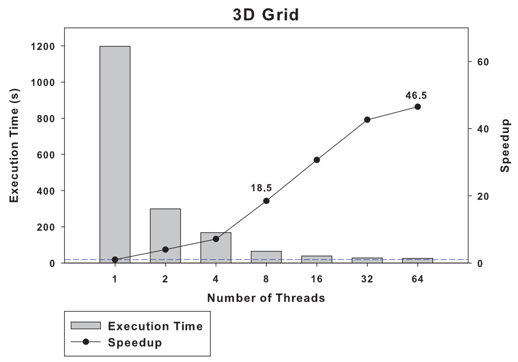


Figure 19: Execution time and speedup for the 3-D grid graph (3.38M vertices, 10.1M edges).

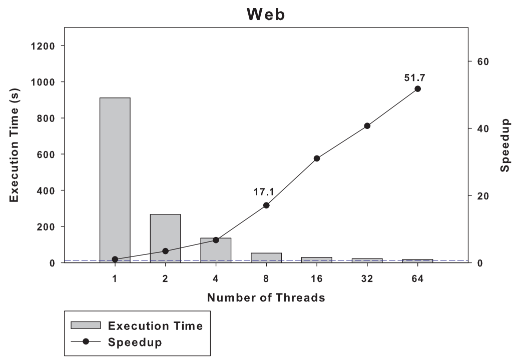


Figure 20: Execution time and speedup for the web graph (3M vertices, 6M edges).

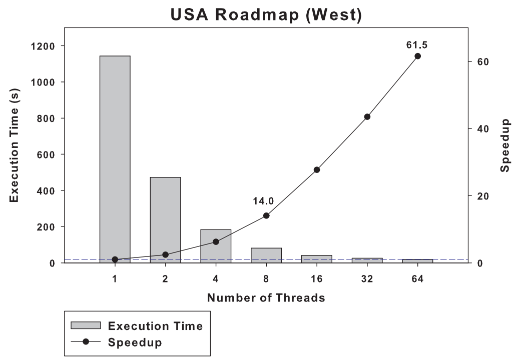


Figure 21: Execution time and speedup for the USA West roadmap graph (6.26M vertices, 7.62M edges).

(18.5 in the best case) for 8 cores for all the test graphs and demonstrated remarkable speedup up to 64 HTs for all the test graphs except for the random graph. Super-linear speedup is achieved by reduced average heap size and the color filtering as expected

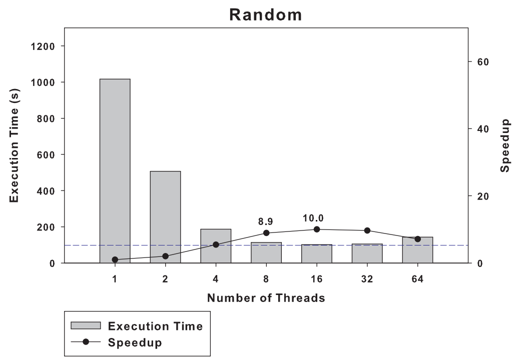


Figure 22: Execution time and speedup for the random graph (3M vertices, 90M edges).

in Section 3.3. For the graphs with a relatively large diameter (the 2-D grid, the web graph, and the USA West graph), our algorithm exhibits smaller numbers of conflicts, which lead to the high scalability. For the graphs with a smaller diameter (the 3-D grid and the random graph as the worst case), the number of conflicts increases but the color filtering compensates for this increase. Our implementation demonstrates remarkable speedup for the 2-D grid, the 3-D grid, the web graph, and the USA West graph. Even for the random graph which involves more frequent data mergings, we achieve parallel speedup using up to 16 threads. The lower scalability after 8 threads in the random graph case is also affected by higher spatial locality; the random graph has higher average vertex degree, which leads to higher spatial locality in accessing neighbor arrays, and in turn, reduces the effectiveness of hardware threads. *However, even with this level of scalability, our parallel algorithm runs only at the comparable speed to the single-threaded case which does not incur STM overhead.*

3.5 Limitations of STM and Requirements for HTM

Even though our STM implementation demonstrates remarkable scalability, the high overhead of the STM system nullifies the speedup. A single memory read or write operation outside a transaction is translated to a single LOAD or STORE instruction. A single shared data read or write operation inside a STM transaction, in contrast,

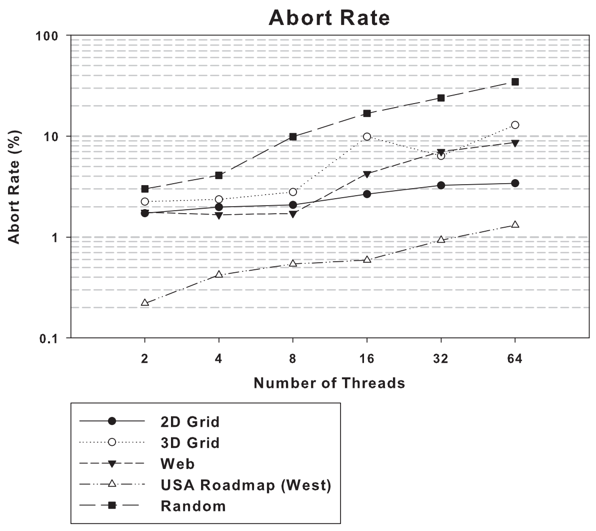


Figure 23: Abort rate for varying numbers of threads.

involves multiple checks and data structure accesses, and this requires a significantly larger number of instructions. Initial bookkeeping and commit time overhead exacerbates the situation. This overhead is not acceptable if we execute a large fraction of the code inside transactions. Ironically, if we execute only a small fraction of the code inside transactions, the single lock approach will suffice to achieve scalability, and we may not need Transactional Memory. This reveals the clear limitation of STM.

HTM can change this situation as a single read or write operation inside a HTM transaction requires only one LOAD or STORE instruction in most currently proposed HTM systems. Still, initial bookkeeping or final commit time overhead can increase the cost of transactions, but this can be managed to a moderate level with hardware support. If HTM can realize this low overhead mechanism in commercial microprocessors, then we can replay the high scalability of our algorithm with only moderate overhead assuming sufficient memory bandwidth and scalable memory subsystem. Then, our algorithm can run significantly faster than the best sequential algorithm for irregular sparse graphs to the level that has not yet been demonstrated by others.

One remaining point to check is MST data merging overhead, as it can be underestimated owing to the high STM overhead. In Table 5, we compare the MST data merging time in the 64 thread cases (the sum of MST data merging time for all 64 threads) with the single-threaded execution time without STM overhead, which will be similar to the single-threaded execution time on an efficient HTM system. Based on the comparison, we can identify that MST data merging time will not significantly lower the scalability except for the case of the random graph. Also, we can expect lower abort rates in HTM systems as the execution time for the transactions will account for a smaller fraction of the total execution time owing to lower transactional overhead. Accordingly, there will be fewer concurrent transactions, and this will contribute to lower abort rates than the case of STM (summarized in Figure 23).

Actually, Dice *et al.* [42] confirmed our argument by running our algorithm—with few modifications to work around Rock processor's limitations. They achieved high scalability and low overhead, which translate to high-performance.

3.6 Summary

Multicore and manycore processors are arising as a new paradigm to pursue. However, to fully exploit all the cores in a chip, parallel programming is often required, and the complexity of parallel programming raises a significant concern. Data synchronization is a major source of this programming complexity, and Transactional Memory has been proposed to reduce the difficulty caused by data synchronization requirements, while providing high scalability and low performance overhead.

The previous literature on Transactional Memory mostly focuses on architectural designs. Its impact on algorithms and applications has not yet been studied thoroughly. In this chapter, we investigate Transactional Memory from algorithm designers' perspective. This chapter presents an algorithmic model to assist the design of efficient Transactional Memory algorithms and a novel Transactional Memory algorithm for computing a minimum spanning forest of sparse graphs. We emphasize multiple Transactional Memory related design issues in presenting our algorithm. We also provide experimental results on an existing software Transactional Memory system. Our algorithm demonstrates excellent scalability in the experiments, but at the same time, the experimental results reveal the clear limitation of software Transactional Memory due to its high performance overhead. Based on our experience, we highlight the necessity of efficient hardware support for Transactional Memory to realize the potential of the technology.

CHAPTER IV

ACCELERATING JPEG2000 STILL IMAGE ENCODING USING THE IBM CELL BROADBAND ENGINE

This paper discusses the optimization of JPEG2000 still image encoding for the IBM Cell Broadband Engine. This chapter is based on *Seunghwa Kang and David A. Bader, "Optimizing JPEG2000 Still Image Encoding on the Cell Broadband Engine," The 37th International Conference on Parallel Processing (ICPP), Portland, OR, September 8th-12th, 2008* and a part of *David A. Bader, Virat Agarwal, and Seunghwa Kang, "Computing Discrete Transforms on the IBM Cell Broadband Engine," Parallel Computing, 35(3):119-137, 2009.*

JPEG2000 [60] is the latest still image coding standard issued by the JPEG committee, which supports both lossless and lossy compression with superior image quality in a low bit rate and additional new features. JPEG2000 adopts Embedded Block Coding with Optimized Truncation (EBCOT) [120] and Discrete Wavelet Transform (DWT) [27, 122] as key algorithms. The EBCOT algorithm consists of three steps: bit modeling, arithmetic coding, and tag tree building. JPEG2000 executes the EBCOT algorithm in two tiers, Tier-1 and Tier-2. Bit modeling and arithmetic coding are performed in Tier-1, whereas tag tree building is performed in Tier-2. Prior analyses of JPEG2000 execution time [74, 6, 80] revealed that Tier-1 coding in the EBCOT and the DWT are the most computationally expensive algorithmic kernels.

The Cell Broadband Engine (or the Cell/B.E.) has unique architectural features with a simple core design and an alternative memory subsystem. The Cell/B.E. chip consists of two types of cores, one PPE and eight SPEs. The PPE is a power efficient version of the PowerPC architecture, and the SPE is a SIMD accelerator. The SPE lacks dynamic branch prediction and runtime out-of-order execution support. It has 256 KB local memory called Local Store. Data transfers between main memory and Local Store require explicit DMA instructions.

Previously, Muta et al. optimized Motion JPEG2000 encoder on the Cell/-B.E. [92]. Motion JPEG2000 encoding lacks inter-frame compression and is nearly identical to JPEG2000 still image encoding, and the authors optimized the DWT and EBCOT algorithms. However, their DWT implementation did not scale beyond a single SPE despite having high single SPE performance. Their EBCOT implementation showed better scalability but did not scale above a single Cell/B.E. processor. This suggests that we need to take a different approach.

Our Cell/B.E. JPEG2000 library is based on Jasper [6], which is a JPEG2000 still image transcoder. Jasper was previously parallelized by Meerwald et al. [80] using OpenMP. However, the authors parallelized Tier-1 coding in the EBCOT and the DWT only to minimize code modification. The maximum achievable speedup was limited by the sequentialization in this loop-level parallelization approach.

We analyze the whole code to investigate the existing concurrency in JPEG2000 and parallelize the level shift stage, the inter-component transform stage, the quantization stage, and a portion of the stream I/O routine in addition to the Tier-1 encoding and DWT stages. We also use a novel data decomposition scheme (discussed in Section 4.1) to achieve high performance while reducing the programming complexity, and we apply the scheme to multiple algorithmic kernels in JPEG2000.

The DWT is one of the most computationally intensive parts in JPEG2000 and also an important kernel in other application areas [97]. This chapter provides a detailed analysis of its performance on the Cell/B.E. We optimize and tune the column grouping strategy based on our data decomposition scheme to increase spatial locality and design a new loop merging approach to increase temporal locality. In addition, we investigate the relative performance of the floating point operations and its fixed point approximation [6] on the Cell/B.E.

We achieve an overall speedup of 6.6 and 3.1 for lossless and lossy encoding with 8 SPEs compared to the single SPE performance. Also, our implementation obtains 6.9 and 7.4 times higher performance over the PPE-only case. The Cell/B.E. demonstrates 3.2 and 2.7 times faster encoding time relative to the Intel Pentium IV 3.2 GHz processor for the lossless and lossy cases, respectively. For the DWT, the Cell-/B.E. outperforms the Pentium IV processor by 9.1 and 15 times for the lossless and lossy cases, respectively. We further test our implementation on a single IBM QS20 blade with two Cell/B.E. processors. The performance scaled up to 16 SPEs. We obtain significantly higher performance than the previous Motion JPEG2000 encoder implementation [92] for the Cell/B.E. as well. Also, we compare the performance of the Cell/B.E. with the AMD Barcelona (Quad-core Opteron) processor for the DWT. We apply various optimization techniques and use PGI C compiler for the Barcelona processor, which enable the head to head comparison of the Cell/B.E. with the general purpose multicore prcoessor. The source code of this work is freely available from our CellBuzz project in SourceForge (`http://sourceforge.net/projects/cellbuzz`).

4.1 Data Decomposition Scheme

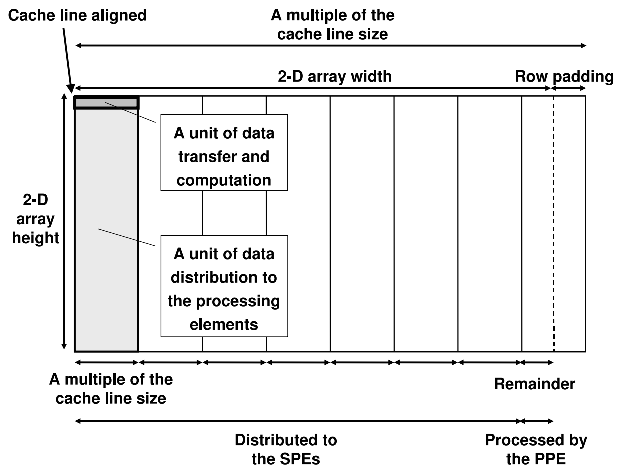

Cache line aligned
A multiple of the cache line size
2-D array width
Row padding
A unit of data transfer and computation
2-D array height
A unit of data distribution to the processing elements
A multiple of the cache line size
Remainder
Distributed to the SPEs
Processed by the PPE

Figure 24: Data decomposition scheme for two dimensional array

Data layout is an important design issue in parallel programming. It is even more important for the Cell/B.E. due to the alignment and size requirements for DMA data transfer and SIMD load/store. DMA on the Cell/B.E. requires 1, 2, 4, 8 byte alignment to transfer 1, 2, 4, 8 bytes of data and 16 byte alignment to transfer a multiple of 16 bytes. DMA data transfer becomes most efficient if the data addresses are cache line aligned in both main memory and SPE Local Store, and the data transfer size is an even multiple of the cache line size [69]. SIMD load/store instructions for vectorization also require quad-word data alignment.

Our data decomposition scheme, shown in Figure 24, satisfies the above requirements for two dimensional arrays with an arbitrary width and height, assuming that every row can be arbitrarily partitioned into multiple chunks for independent processing. First, our data decomposition scheme pads every row to force the start address of every row to be cache line aligned. Then, the scheme partitions the data array to multiple chunks, and every chunk except for the last has the width a multiple of the cache line size. All the chunks have the height identical to the data array height. These data chunks become a unit of data distribution to the processing elements. The constant width chunks are distributed to the SPEs, and the PPE processes the last remainder chunk. The SPE traverses the assigned chunks by processing every single row in the chunk as a unit of data transfer and computation.

This data decomposition scheme has several impacts on the performance and the programmability. First, every DMA transfer in the SPE becomes cache line aligned, and the transfer size becomes a multiple of the cache line size. This results in efficient DMA transfers and reduced programming complexity. If data are not properly aligned, or the data transfer size has an arbitrary value, additional programming would have been required to satisfy the conditions for correctness. Reduced programming complexity, or in other words, the shorter code size also saves the Local Store space, and this is important for the Cell/B.E. since the Local Store size is relatively small. Also, under our data decomposition scheme, there is no cache conflict since every cache line in the data array is accessed either by the PPE or a DMA instruction issued by one SPE. The remainder chunk with an arbitrary width is processed by the PPE to enhance the overall chip utilization. Our data decomposition scheme also satisfies the alignment requirement for SIMD load/store.

Second, the Local Store space requirement becomes constant independent of data array size. As mentioned above, a single row in a chunk, which has the constant width, becomes a unit of data transfer and computation in the SPE. This leads to a constant memory requirement, and we can easily adopt optimization techniques that require additional Local Store space. For example, double buffering or multi-level buffering is an efficient technique for hiding latency but increases the Local Store space requirement at the same time. However, owing to the constant memory requirement in our data decomposition scheme, we can increase the level of buffering to a higher value that fits within the Local Store.

In addition, fixed data size leads to a constant loop count, which enables compilers to better predict the program's runtime behavior. This helps compilers to use optimization techniques such as loop unrolling, instruction rescheduling, and compile time branch prediction. Compile time instruction rescheduling and branch prediction compensate for the lack of runtime out-of-order execution and dynamic branch prediction support in the SPE.

4.2 Parallelization of JPEG2000

4.2.1 Parallelism in JPEG2000

The level shift, inter-component transform, and quantization stages are basically pixel-wise independent. Therefore, arbitrary partitioning is possible. Discrete Wavelet Transform (DWT) consists of two steps. First, the vertical filtering step partitions the original image to the low pass subband and the high pass subband in the vertical direction, and then the horizontal filtering step is followed to partition in the horizontal direction. In the vertical filtering, every column can be processed independently, and in the horizontal filtering, every row can be processed independently. For Tier-1 encoding in the EBCOT algorithm, an image array is partitioned to multiple code blocks, and every code block can be processed independently.

4.2.2 Parallelization Strategy

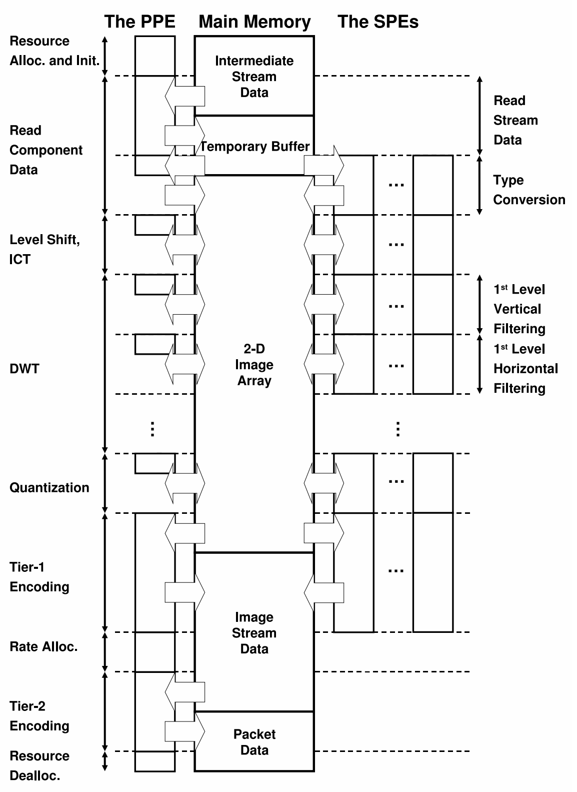


Figure 25: Work partitioning among the PPE and the SPEs for JPEG2000 encoding

Figure 25 summarizes the work partitioning among the PPE and the SPEs for JPEG2000 encoding. We fully parallelize the level shift, inter-component transform, DWT, Tier-1 encoding and quantization stages using both the PPE and the SPEs. We also merge the level shift and inter-component transform stages to minimize the data transfer. We partially parallelize the read component data stage, which includes type conversion from the Jasper specific intermediate data type to the four byte integer data type. We apply our data decomposition scheme to every parallelized stage except for the Tier-1 encoding stage. For the Tier-1 encoding stage, we adopt a work queue to distribute the workloads to the processing elements.

In Tier-1 encoding, efficient DMA data transfer is less important owing to the relatively high computation to communication ratio. Thus, we focus on increasing the overall chip utilization while minimizing the interaction among the processing elements. Our code uses the PPE and SPE threads to encode the code blocks and maintains a work queue for load balancing. Note that the processing time for Tier-1 encoding is dependent on the input data characteristics, and we cannot achieve load balancing by merely distributing an identical number of code blocks to the processing elements. Both our implementation and the previous Motion JPEG2000 encoder [92] use a work queue for load balancing but the two use the PPE for different purposes. In [92], the authors implemented lossless encoding only and overlapped the Tier-1 encoding stage with the Tier-2 encoding stage. In their implementation, the PPE (or the PPEs in their two Cell/B.E. implementations) performs Tier-2 encoding and the code block distribution to the SPEs, while only the SPE threads perform Tier-1 encoding. In the lossy encoding process, the rate control stage appears between the Tier-1 encoding stage and the Tier-2 encoding stage. This prevents the overlap in our implementation, and our code uses the PPE and SPE threads for Tier-1 encoding. Another distinction is the code block size. In [92], the authors selected 32 by 32 pixels code block size instead of 64 by 64, the maximum code block size in the standard. Smaller code block size reduces the Local Store memory requirements and enables double buffering, but increases the interaction among the PPE and SPE threads and lowers scalability. We use 64 by 64 in favor of scalability.

Previous parallelization strategies for the DWT [129, 97, 31] involve trade-offs between computation and communication overhead for different underlying architectures. Sweldens [117] proposed a lifting based DWT, by which, an in-place DWT can be performed faster than the previous convolution based DWT [70]. Still, poor cache behavior in a column-major traversal in a C language implementation becomes a bottleneck in performance. Initially, a matrix transpose, which converts a column-major traversal to a row-major traversal, was adopted to improve cache behavior. Chaver *et al.* introduced loop tiling (or column grouping) to improve cache behavior without matrix transpose. In [92], the authors parallelized the convolution based DWT for the Cell/B.E. by partitioning the data array to 128 by 128 pixels tiles with the overlap among the adjacent tiles. The authors used the net tile size of 112 by 112 pixels, which does not satisfy the cache line alignment requirements for the most efficient DMA transfer due to the overlapped area. We optimize the lifting based DWT for the Cell/B.E. For the horizontal filtering, our code assigns an identical number of rows to each SPE, and a single row becomes a unit of data transfer and computation. Every row is cache line aligned, and the data transfer size becomes a multiple of the cache line size owing to the row padding in our data decomposition scheme. For the vertical filtering, we tune the column grouping approach by fixing column group size to a multiple of the cache line size to use our data decomposition scheme. Our implementation enhances DMA data transfer efficiency, which is essential to achieve high scalability.

4.3 Vectorization of JPEG2000

Based on our data decomposition scheme, we vectorize the level shift, inter-component transform, and quantization stages in a straightforward way. Yet, vectorization of the DWT involves interesting issues related to the Cell/B.E., and we discuss the issues in this section.

In [6], the authors suggested the fixed point representation for the real numbers in the JPEG2000 lossy encoding process to enhance the performance and the portability. The authors assumed that fixed point instructions are generally faster than floating

Table 6: Latency for the SPE instructions

Instruction	Description	Latency
mpyh	two byte integer multiply high	7 cycles
mpyu	two byte integer multiply unsigned	7 cycles
a	add word	2 cycles
fm	single precision floating point multiply	6 cycles

point instructions. However, the current version of the Cell/B.E. chip is optimized for (single precision) floating point operations, and the floating point instructions have comparable speed to the fixed point instructions. Moreover, the SPE instruction set architecture does not support four byte integer multiplications; thus four byte integer multiplications need to be emulated by two byte integer multiplications and additions. Table 6 summarizes the latency for the two byte integer multiplication, four byte integer addition, and single precision floating point multiplication. Therefore, the fixed point representation loses its benefit on the Cell/B.E. We replace the fixed point representation in Jasper code with the floating point representation to achieve high performance.

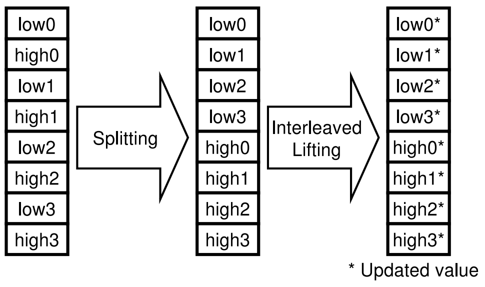


Figure 26: The splitting step and the interleaved lifting step for the vertical filtering

The DWT algorithm consists of multiple steps: one splitting step and two lifting steps in the lossless mode and one splitting step, four lifting steps, and one optional scaling step in the lossy mode. Considering that the entire column group data for a large image does not fit into the Local Store, 3 or 6 steps in the vertical filtering involve 3 or 6 DMA data transfers of the entire column group data. As the number of SPEs

```
Input: an image data array $array\_data$, number of rows $number\_of\_rows$, row width
       including padding $stride$
Output: an image data array $array\_data$
$high\_start \leftarrow number\_of\_rows/2$;
/* 1st lifting step for the vertical filtering */
$p\_low \leftarrow array\_data[0]$;
$p\_high \leftarrow array\_data[high\_start * stride]$;
$n \leftarrow high\_start - 1$;
for $i \leftarrow 0$ to $n - 1$ do
    $*p\_high \leftarrow *p\_high - ((*p\_low + *(p\_low + stride))/2)$;
    $p\_low \leftarrow p\_low + stride$;
    $p\_high \leftarrow p\_high + stride$;
end
$*p\_high \leftarrow *p\_high - *p\_low$;
/* 2nd lifting step for the vertical filtering */
$p\_low \leftarrow array\_data[0]$;
$p\_high \leftarrow array\_data[high\_start * stride]$;
$*p\_low \leftarrow *p\_low + ((*p\_high + 1)/2)$;
$p\_low \leftarrow p\_low + stride$;
$n \leftarrow high\_start - 1$;
for $i \leftarrow 0$ to $n - 1$ do
    $*p\_low \leftarrow *p\_low + ((*p\_high + *(p\_high + stride) + 2)/4)$;
    $p\_low \leftarrow p\_low + stride$;
    $p\_high \leftarrow p\_high + stride$;
end
```

Algorithm 2: Pseudo code for the original implementation

increases, the limited off-chip memory bandwidth becomes a bottleneck and nullifies the performance enhancement achieved by vectorization. To reduce the amount of DMA data transfer, we interleave the lifting steps. Algorithm 2 illustrates the original algorithm for the lossless mode assuming that the number of rows is even and larger than four. By analyzing the data dependency in Algorithm 2, we notice that two lifting steps can be interleaved. Algorithm 3 depicts our interleaved algorithm. The splitting step can also be merged with the next two lifting steps. Figure 26 illustrates the splitting step and the interleaved lifting step in the lossless vertical filtering. Initially, the low and high pass components are interleaved in the input array, and the splitting step separates the low and high pass components. This splitting step

```
Input: an image data array $array\_data$, number of rows $number\_of\_rows$, row width
       including padding $stride$
Output: an image data array $array\_data$
$high\_start \leftarrow number\_of\_rows/2$;
/* interleaves 1st and 2nd lifting steps for the vertical filtering */
$p\_low \leftarrow array\_data[0]$;
$p\_high \leftarrow array\_data[high\_start * stride]$;
$*p\_high = *p\_high - ((*p\_low + *(p\_low + stride))/2)$;
$*p\_low = *p\_low + ((*p\_high + 1)/2)$;
$p\_low \leftarrow p\_low + stride$;
$p\_high \leftarrow p\_high + stride$;
$n \leftarrow high\_start - 2$;
for $i \leftarrow 0$ to $n - 1$ do
    $*p\_high \leftarrow *p\_high - ((*p\_low + *(p\_low + stride))/2)$;
    $*p\_low \leftarrow *p\_low + ((*(p\_high - stride) + *p\_high + 2)/4)$;
    $p\_low \leftarrow p\_low + stride$;
    $p\_high \leftarrow p\_high + stride$;
end
$*p\_high \leftarrow *p\_high - *p\_low$;
$*p\_low \leftarrow *p\_low + ((*(p\_high - stride) + *p\_high + 2)/4)$;
```

Algorithm 3: Pseudo code for our interleaved implementation

involves the DMA data transfer of the whole column group data. If we adjust the pointer addresses for the low and high pass components and increase the increment size in the next interleaved lifting step, the splitting step can be merged with the interleaved lifting step. This merges three steps in the lossless mode to a single step and reduces the amount of data transfer. However, updating the high pass part in the merged single step overwrites the input data before it is read, and this leaves us with a problem. In Figure 26, *high0* and *low0* are updated first in the interleaved lifting step, and *high1* and *low1* are updated next. If we adjust the input data pointer and skip the splitting step, updating *high0* overwrites *low2* in the input data array before it is read. To remedy this problem, our code uses an auxiliary buffer (in main memory), and the updated high pass data are written to the buffer first and copied to the original data array after the single merged step is finished. The amount of data transfer related to the auxiliary buffer is half of the entire column group, and this halves the amount of data transfer for the splitting step. We recently find that a similar idea appears in [72] for the lossy case. Loop fusion for the four lifting steps is described in the paper. By combining this idea with our approach, we merge one splitting step, four lifting steps, and the optional scaling step in the lossy mode into a single loop to further reduce the DMA bandwidth requirement.

4.4 Experimental Results

We use the *gcc* compiler in the Cell SDK 2.1 with *-O5* optimization flag for the performance analysis. Our baseline code is the open source Jasper 1.900.1 (`http://www.ece.uvic.ca/~mdadams/Jasper`), and we use a 28.3 MB 3800 × 2600 color image as a test file. The default option and *-O mode=real -O rate=0.1* are applied for lossless and lossy encoding, respectively. We transcode the image from the BMP format to the JPEG2000 format and record the JPEG2000 encoding time. We disregard the BMP decoding time. Our experiments use an IBM QS20 Cell blade server with dual Cell/B.E. 3.2 GHz chips (rev. 3) and 1 GB main memory.

4.4.1 Encoding Time and Scalability

Figures 27 and 28 display execution time and speedup for lossless and lossy encoding, respectively. As the EBCOT algorithm is branchy and integer based, the PPE runs the code faster than the SPE for Tier-1 encoding. Therefore, the 1 PPE only case outperforms the single SPE's performance. Still, we achieve near linear speedup with increasing numbers of SPEs and the extra speedup with additional PPE threads. As a consequence, the 16 SPE + 2 PPE case completes Tier-1 encoding significantly faster than the 1 PPE only case.

In the case of the DWT, the 1 SPE case outperforms the 1 PPE only case by far, and we demonstrate a remarkable speedup with additional SPEs. Vectorization contributes to the superb single SPE performance, and in the lossy encoding case, the execution time is further reduced by replacing the fixed point representation with

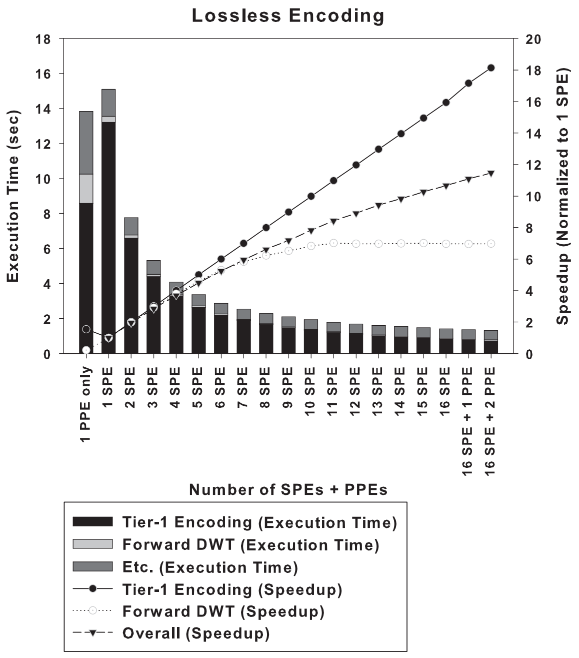


Figure 27: Execution time and speedup for lossless encoding. Additional PPEs participate in Tier-1 encoding.

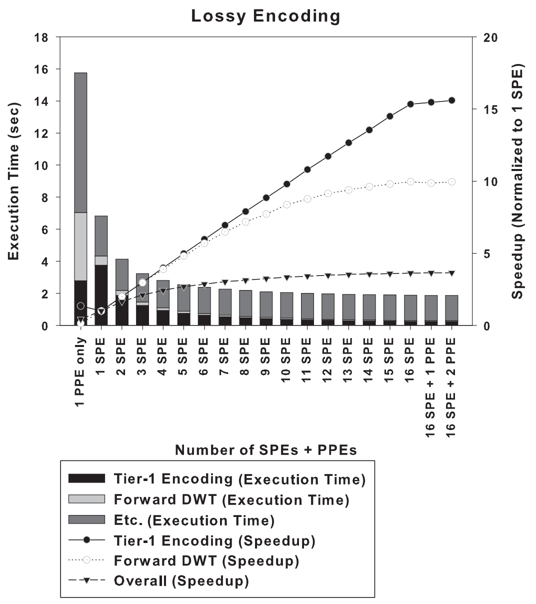


Figure 28: Execution time and speedup for lossy Encoding. Additional PPEs participated in Tier-1 encoding.

the floating point representation. The efficient use of the off-chip memory bandwidth based on our data decomposition scheme and the reduced bandwidth requirement owing to the loop interleaving realize the high scalability.

Overall lossless encoding demonstrates a parallel speedup while the lossy encoding performance flattens with increasing numbers of SPEs due to the sequential rate allocation stage, which accounts for approximately 60% of the total execution time in the 16 SPE + 2 PPE case.

4.4.2 A Comparison with the Previous Implementation

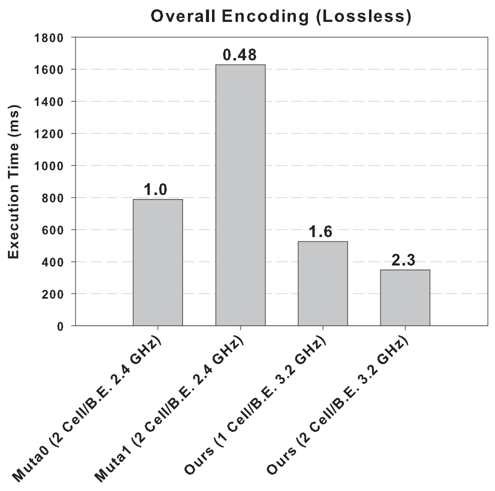


Figure 29: An overall performance comparison with the previous implementations for the Cell/B.E. The numbers above the bars denote the speedup relative to Muta0.

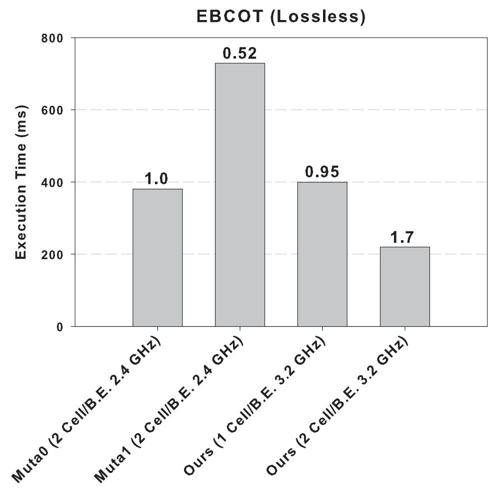


Figure 30: An EBCOT (Tier-1 + Tier2) encoding performance comparison with the previous implementations for the Cell/B.E. The numbers above the bars denote the speedup relative to Muta0.

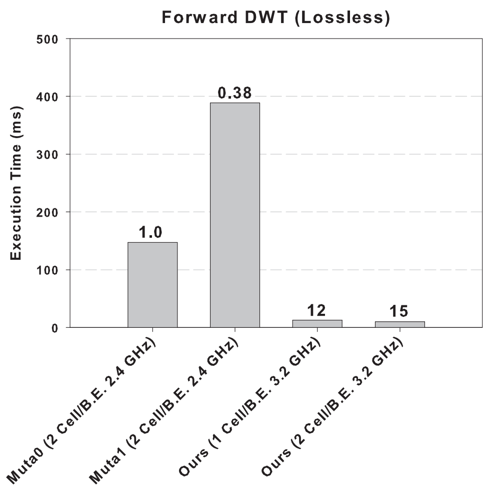


Figure 31: A DWT performance comparison with the previous implementations for the Cell/B.E. The numbers above the bars denote the speedup relative to Muta0

We compare the performance of our code with Muta *et al.*'s implementations [92]. Figures 29, 30, and 31 summarize the performance comparison. Muta0 and Muta1 in the figures denote the implementations in [92]. In Muta0, two encoding threads encoded two different frames concurrently with two Cell/B.E. processors. The two chips worked in a synergistic way to increase the overall throughput. In Muta1, one encoding thread encoded all the frames using two Cell/B.E. processors.

We use the performance numbers reported by the authors for the comparison. The authors excluded the Motion JPEG2000 format building time and measured the encoding time for 24 frames with the size of 2048 × 1080. Total execution time was divided by the number of encoded frames to compute per frame encoding time. In Muta0, the encoding time for one frame can be up to two times higher than the reported number. We scale down the test image for our implementation to 2048×1080 for fair comparison.

Still, there are multiple caveats in the comparison. First, the Cell/B.E. 2.4GHz, instead of the Cell/B.E. 3.2 GHz, was used in [92]. Second, the experimental results for our implementation based on Jasper include reading and type conversion time from the Jasper specific intermediate stream, JPEG2000 format building time, and final file I/O time to save the encoded file. This may be different from [92]. Third, even though the test images have the identical size, different characteristics of the images may affect the result.

Our implementation with one Cell/B.E. processor and two Cell/B.E. processors demonstrates superior overall performance than the previous implementations with the two Cell/B.E. processors. This is mainly due to the following reasons. First, our EBCOT implementation demonstrates higher scalability than the previous implementation. Minimized communication between the PPE and the SPEs enhances the scalability in addition to the implementation details. Second, adopting the lifting based scheme instead of the convolution based scheme, combined with the higher chip clock frequency, results in the higher single SPE performance for the DWT, and our data decomposition scheme and the loop interleaving contribute to the higher scalability. Third, we parallelized the level shift, inter-component transform, and quantization stages while these stages were executed on the PPE in [92] to avoid the offloading overhead. The offloading overhead in our implementation is insignificant owing to the effective data decomposition scheme.

4.4.3 A Comparison with the Intel x86 Architecture

Figure 32 summarizes a performance comparison between the Cell/B.E. and the Intel Pentium IV 3.2 GHz with 2 MB cache memory and 2 GB main memory. Jasper code on the Pentium IV processor is compiled with the *gcc* version 4.1.2 and *-O5* optimization flag (instead of the default *-g -O2*). To make the comparison as fair as possible, we apply the optimizations except for parallelization, vectorization, and other optimizations specific to the Cell/B.E. to both architectures. Note that the Pentium IV processor also supports SIMD instructions, but vectorization is not implemented in the Jasper code for the Pentium IV processor. Also, for lossy encoding, the Cell/-B.E. performs the floating point arithmetic while the Pentium IV processor emulates

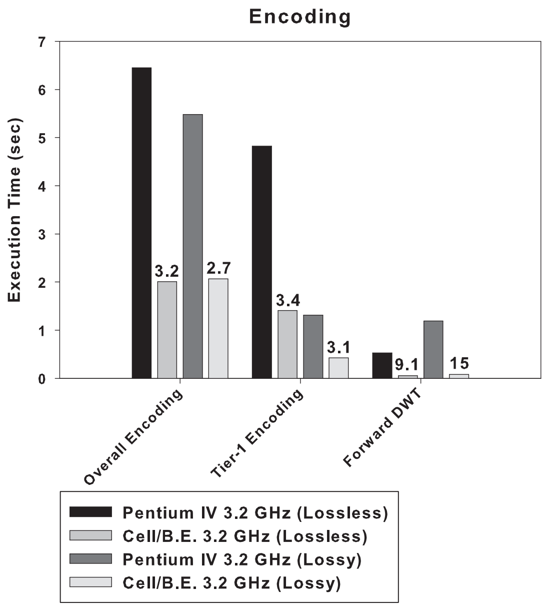


Figure 32: An encoding performance comparison of the Cell/B.E. to that of the Intel Pentium IV 3.2 GHz processor. The numbers above the bars denote the speedup relative to the Intel Pentium IV processor.

the floating point operations with the fixed point instructions. The Cell/B.E. outperforms the Pentium IV in the comparison. Especially, the Cell/B.E. demonstrates the impressive performance for the DWT while the sequential part of the code running on the PPE lowers the overall speedup. These performance numbers show that the Cell/B.E. has superb performance for the floating point based loop intensive algorithms, and its relatively low single core performance for the branchy and integer based algorithms can be compensated by exploiting multiple SPEs.

We also optimize the DWT code for the AMD Barcelona for the comparison. We optimize the code for Barcelona using PGI C compiler, a parallel compiler for multicore optimization, and user provided compiler directives. The following summarizes the performance optimizations applied.

- **Parallelization:** OpenMP based parallelization.
- **Vectorization:** Auto-vectorization with compiler directives for pointer disambiguation.

- **Real number representation:** Identical to the Cell/B.E. case.
- **Loop merging:** Identical to the Cell/B.E. case.
- **Run-time profile feedback:** Compile with the run-time profile feedback (-Mpfo).

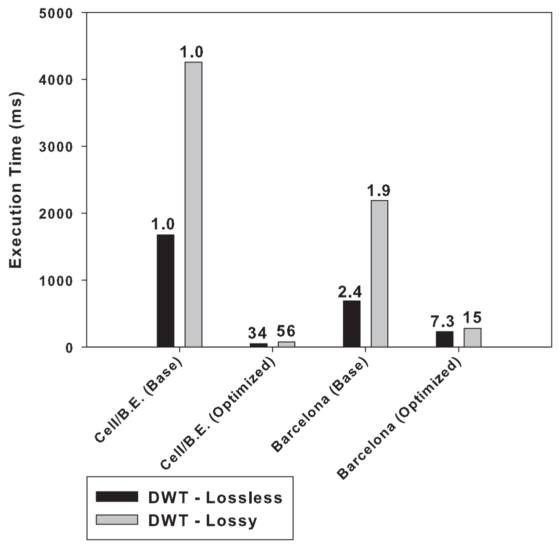


Figure 33: A DWT encoding performance comparison of the Cell/B.E. to that of the AMD Barcelona (Quad-core Opteron) processor. The numbers above the bars denote the speedup relative to the baseline implementation on the Cell/B.E.

Figure 33 summarizes the performance comparison between the Cell/B.E. and the AMD Barcelona 2.0 GHz (Quad-Core Opteron Processor 8350) for the DWT part. The above result shows that the baseline implementation for the Cell/B.E. (running on the PPE only) runs slower, but the Cell/B.E. optimized code runs significantly faster than the AMD Barcelona optimized code. This demonstrates the Cell/B.E. processor's performance potential in processing regular and communication-intensive applications. This also reveals the importance of the architecture-aware algorithm design and tuning as well as the performance potential of emerging accelerator architectures.

4.5 Summary

JPEG2000 is the latest still image coding standard from the JPEG committee, which adopts new algorithms such as Embedded Block Coding with Optimized Truncation (EBCOT) and Discrete Wavelet Transform (DWT). These algorithms enable superior coding performance over JPEG and support various new features at the cost of the increased computational complexity. The Sony-Toshiba-IBM Cell Broadband Engine (or the Cell/B.E.) is a heterogeneous multicore architecture with SIMD accelerators. In this work, we optimize the computationally intensive algorithmic kernels of JPEG2000 for the Cell/B.E. and also introduce a novel data decomposition scheme to achieve high performance with low programming complexity. We compare the Cell/B.E.'s performance to the performance of the Intel Pentium IV 3.2 GHz processor (for the entire encoding process) and also the AMD Barcelona architecture (for the DWT). The Cell/B.E. demonstrates 3.2 times higher performance for lossless encoding and 2.7 times higher performance for lossy encoding. For the DWT, the Cell/B.E. outperforms the Pentium IV processor by 9.1 times for the lossless case and 15 times for the lossy case. The Cell/B.E. also outperforms the Barcelona in our experiments. We also provide the experimental results on an IBM QS20 blade with two Cell/B.E. chips and the performance comparison with the existing JPEG2000 encoder for the Cell/B.E.

CHAPTER V

LARGE SCALE COMPLEX NETWORK ANALYSIS USING THE HYBRID COMBINATION OF A MAPREDUCE CLUSTER AND A HIGHLY MULTITHREADED SYSTEM

This chapter analyzes major computational challenges in large scale complex network analysis and suggests a hybrid system to address the challenges. This chapter is based on the materials in *Seunghwa Kang and David A. Bader, "Large Scale Complex Network Analysis using the Hybrid Combination of a MapReduce cluster and a Highly Multithreaded System," The 4th Workshop on Multithreaded Architectures and Applications (MTAAP), Atlanta, Georgia, April 23rd, 2010.*

Complex Networks [38] abstract interactions among entities in a wide range of domains—including sociology, biology, transportation, communication, and the Internet—in a graph representation. Analyzing these complex networks solves many real-world problems. Watts and Strogatz [127] found that a small graph diameter, which is a common feature in many complex networks [127], has a significant impact on the spread of infectious diseases. Albert *et al.* [8] studied the impact of a power-law degree distribution [7, 44] on the vulnerability of the Internet and the efficiency of a search engine. Bader and Madduri [17] computed the betweenness centrality of protein-interaction networks to predict the lethality of the proteins, and Madduri *et al.* [77] applied an approximate betweenness centrality computation to the IMDb movie-actor network and found important actors with only a small number of connections.

Analyzing large scale complex networks, however, imposes difficult computing challenges. Graphs that represent complex networks commonly have millions to billions of vertices and edges. These graphs are often embedded in raw (and often streaming) data—*e.g.* web documents, e-mails, and published papers—of terabytes to petabytes. Extracting a compact representation of a graph or subgraph from large volumes of data is a significant challenge. The irregular structure of these graphs stresses traditional hierarchical memory subsystems as well. These graphs also tend not to partition well for multiple computing nodes; the partitioning of these graphs has significantly larger edge cuts than the partitioning of traditional graphs which are derived from physical topologies [18]. This necessitates large volumes of inter-node communication. In this chapter, we study a problem of extracting a subgraph from a larger graph—to capture the challenge in processing large volumes of data—and finding single-pair shortest paths in the subgraph—to capture the challenge in irregularly traversing complex networks.

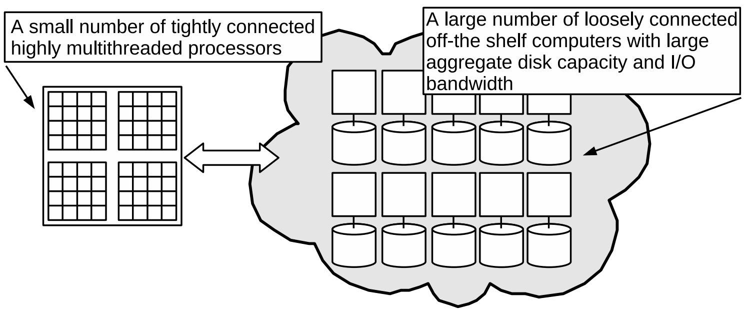

A small number of tightly connected highly multithreaded processors
A large number of loosely connected off-the shelf computers with large aggregate disk capacity and I/O bandwidth

Figure 34: The hybrid combination of a MapReduce cluster and a highly multithreaded system.

We map our problem onto three different platforms: a MapReduce cluster, a highly multithreaded system, and the hybrid combination of the two. Cloud computing, using the MapReduce programming model, is becoming popular for data-intensive analysis. A Cloud computing system with the MapReduce programming model—or a MapReduce cluster—is efficient in extracting a subgraph via filtering. Finding a single-pair shortest path, however, requires multiple dependent indirections with irregular data access patterns. The MapReduce algorithm to find a single-pair shortest path is not work optimal and also requires large bisection bandwidth to scale on large systems (see Section 5.3). In our experiment, a single-pair shortest path problem runs five orders of magnitude slower on the MapReduce cluster than the highly multithreaded system with a single Sun UltraSparc T2 processor.

A highly multithreaded system—with the shared memory programming model—is efficient in supporting a large number of irregular data accesses all across the memory space. This system, however, often has limited computing power, memory and disk capacity, and I/O bandwidth and inefficient or even impossible to process very large graph data. Finding a single-pair shortest path in the subgraph, in contrast, fits well with the programming model and the architecture.

The hybrid system (see Figure 34) exploits the strengths of the two different architectures in a synergistic way. A MapReduce cluster extracts a subgraph, and a highly multithreaded system loads the subgraph from the cluster and finds single-pair shortest paths. The subgraph loading time is less significant if the subgraph size is much smaller than the original data. The hybrid system solves our problem in the most efficient way in our experimentation.

5.1 Complex Network Analysis

Complex networks are often embedded in large volumes of real-world data. Extracting a graph representation from raw data is a necessary step in solving many complex network analysis problems. Analyzing subgraphs of a larger graph is also an interesting problem. Costa *et al.* [38] surveyed measurement parameters for complex networks and presented several examples in this direction. Subgraphs with the edges created in different time intervals along the growth of a network reveal the dynamic characteristics of the network. Rich-club coefficient measures interactions among only highly influential entities—or high degree vertices using terminologies in graph theory. Filtering only a certain type of vertices—*e.g.* finding the network of Atlantans or computer scientists in the larger Facebook network—also creates interesting subgraphs to investigate. A common operation involved in the above cases is filtering large input data. A large part of the raw data is unnecessary in generating a graph representation. Extracting a subgraph involves scanning large volumes of data and filtering out vertices and edges that are not part of the subgraph.

The size of an extracted graph is often significantly smaller than the size of the input data, and the memory requirement is less demanding with the sparse graph representation—*i.e.* the adjacency list format which stores the list of neighbors for every vertex. The sparse graph representation requires $O(n + m)$ memory space—for n vertices and m edges—with a small hidden constant. Graphs with multiple millions to billions of edges fit into the main memory of moderate size systems—*e.g.* Madduri *et al.* [77] computed the approximate betweenness centrality of a graph with 134 million vertices and 1.07 billion edges using a Sun T5120 server with 32 GB of DRAM. For graphs represented in the sparse representation, a large fraction of practical complex networks and subnetworks is likely to fit into the memory capacity of moderate size systems, and if this is the case, the match between the graph analysis problems' computational requirements and the architectures' capability becomes more important than the mere capacity.

A Description of Our Problem. We study the problem of extracting a subgraph from a larger graph and finding single-pair shortest paths in the subgraph. We find shortest paths (one shortest path per pair) for up to 30 pairs, which are randomly picked out of the vertices in the subgraph. To extract a subgraph, we filter the input graph to include only the edges that connect the vertices in the subnetwork—we experiment with three subnetworks that cover approximately 10%, 5%, and 2% of the entire vertices—and create adjacency lists from the filtered edges. We also assume that the input graph is generated in advance and stored in the MapReduce cluster.

The input graph is an R-MAT graph [29] with 2^{32} (4.29 billion) vertices and 2^{38} (275 billion) undirected edges. Sampling from a Kronecker product generates an R-MAT graph which exhibits several characteristics similar to social networks such as the power-law degree distribution and the clustering structure. The graph has an order of magnitude more vertices than the Facebook network with a comparable average vertex degree—the Facebook network has over 250 million active users with 120 friends per user in average. We use the R-MAT parameters a=0.55, b=0.1, c=0.1, d=0.25. The graph size is 7.4 TB in the text format.

5.2 MapReduce

A MapReduce cluster is typically composed of a large number of commodity computers connected with off-the-shelf interconnection technologies. The MapReduce programming model frees programmers from the work of partitioning, load balancing, explicit communication, and fault tolerance in using clusters. Programmers provide only map and reduce functions (see Figure 35). Then, the runtime system partitions the input data and spawns multiple mappers and reducers. The mappers apply the map function—which generates an output (key, value) pair—to the partitioned input data. The reducers shuffle and sort the map function output and invoke the reduce function once per each key with a key and the values associated with the key as input arguments.

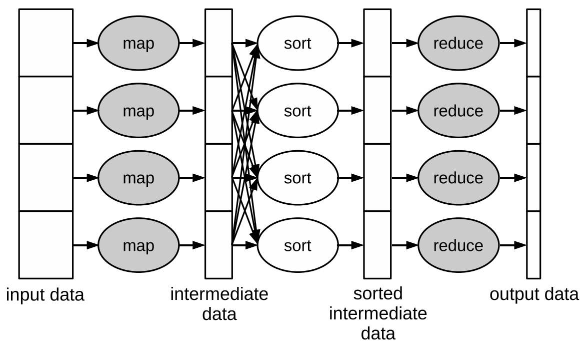


Figure 35: A MapReduce workflow. A programmer provides map and reduce functions (gray shading), and the runtime system is responsible for the remaining parts.

5.2.1 Algorithm Level Analysis

In this section, we provide a criterion to test the optimality of MapReduce algorithms based on the number of MapReduce iterations and the amount of work in each MapReduce iteration. Assume an edge list with m edges. To extract a subgraph from the list, we need to inspect the edges and include only the edges that belong to the subnetwork of interest. This is local computation which only uses data in a single edge representation, and a single invocation of a map function is sufficient to process one edge. Forming adjacency lists from the filtered edges requires global computation which accesses multiple edges at the same time. All edges incident on a same vertex need to be co-located. A MapReduce algorithm co-locates the edges in the shuffle and sort phases via indirections with a key. This is different from $O(1)$ complexity random accesses, or indirections with an address, for the random access machine (RAM) model. One pair of the shuffle and sort phases can serve all independent indirections. As co-locating incident edges for one vertex is independent of co-locating incident vertices for other vertices, a single MapReduce iteration is sufficient for the subgraph extraction.

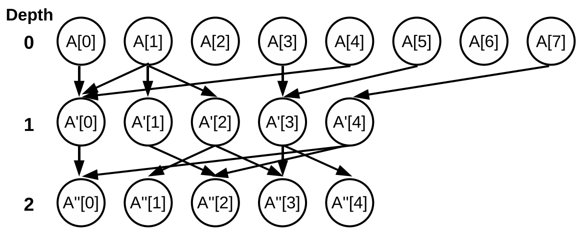


Figure 36: A directed acyclic graph (DAG) for MapReduce computation.

Given a pair of vertices s and t, we find a shortest path from s to t by running two breadth-first searches from s and t until the frontiers of the two meet each other. Each breadth-first search first inspects every vertex one hop away from the source vertex, then visits the vertices one hop away from the neighbors of the source vertex, and so on. This expands the frontier by one hop in each step, and there is dependency between indirections involved in the successive steps. Assume a directed acyclic graph (DAG) is constructed with an element (say $A[i]$) of the input data (say A) as a vertex and indirections as edges (see Figure 36)—one adjacency list of the input adjacency lists is mapped as a vertex in our case. Local computation, which accesses only $A[i]$, is ignored in constructing the DAG. The longest path of dependent indirections determines the depth of the DAG and accordingly the number of required MapReduce iterations. $\lceil d/2 \rceil$, where d denotes the distance from s to t, sets the minimum number of MapReduce iterations to find the shortest path in our problem.

Assume the input data A has n elements, and each element of A is smaller than a certain constant that is much smaller than the input data size. Then, the amount of work in the map, shuffle, sort, and reduce phases and a single MapReduce iteration—W_{map}, $W_{shuffle}$, W_{sort}, W_{reduce}, and $W_{iteration}$, respectively—become the following.

$$\begin{aligned}
W_{map} &= O(n(1+f)) \\
W_{shuffle} &= O(nf) \\
W_{sort} &= p_r \times Sort\left(\frac{nf}{p_r}\right) \\
W_{reduce} &= O(nf(1+r)) \\
W_{iteration} &= O(n + nf + nfr) + p_r \times Sort\left(\frac{nf}{p_r}\right)
\end{aligned}$$

where $f = \frac{map\ output\ size}{map\ input\ size}$, $p_r = a\ number\ of\ reducers$, and $r = \frac{reduce\ output\ size}{reduce\ input\ size}$. Each reducer sorts its input data sequentially, and $Sort(n)$ is $O(n)$—with bucket sort—assuming a finite key space and $O(n\ log\ n)$ otherwise. If solving a problem requires k iterations, the amount of work to solve the problem on a MapReduce cluster, or $W_{MapReduce}$ becomes

$$W_{MapReduce} = \sum_{i=1}^{k} \left(O(n_i + n_i f_i + n_i f_i r_i) + p_r \times Sort\left(\frac{n_i f_i}{p_r}\right)\right)$$

where $n_1 = n$ and $n_{i+1} = n_i f_i r_i$. f_i and r_i are f and r for the ith iteration, respectively. A parallel random access machine (PRAM) algorithm is optimal if $W_{PRAM}(n) = \Theta(T^*(n))$ [61], where W_{PRAM} is the amount of work for the PRAM algorithm and T^* is the time complexity of the best sequential algorithm assuming the RAM model [61]. We extend this optimality criterion for MapReduce algorithms, and a MapReduce algorithm is optimal if $W_{MapReduce}(n) = \Theta(T^*(n))$. Every phase in a MapReduce iteration is trivially parallel, and the time complexity of MapReduce computation to solve a problem with p nodes, or $T_{MapReduce}(n, p)$, is $W_{MapReduce}(n)/p$ assuming $p \ll n$. To realize this time complexity in physical systems, communication in the shuffle phase needs to be minimized and overlapped with the preceding map phase unless bisection bandwidth of the MapReduce cluster grows in proportion to the number of compute nodes.

5.2.2 System Level Analysis

Here, we analyze the architectural features of a MapReduce cluster and their impact on MapReduce algorithms' performance. The MapReduce programming model is oblivious to the mapping of specific computations to specific computing nodes, and the real-world implementations of the programming model—Google MapReduce [40] and open-source Hadoop [4]—provide very limited control over this mapping. Thus, the communication patterns in the shuffle phase are arbitrary, and with arbitrary communication patterns, approximately one half of the map phase output crosses the worst case bisection of a MapReduce cluster. Dividing the map phase output by one half of the bisection bandwidth of a MapReduce cluster calculates the execution time for the shuffle phase, or $T_{shuffle}$ as a result. The shuffle phase, however, can be overlapped with the preceding map phase. Say T_{map} is the execution time for the map phase. Then, the execution time for the map and shuffle phases is $max(T_{map}, T_{shuffle})$. $T_{shuffle}$ does not affect the overall execution time as long as $T_{map} \geq T_{shuffle}$. T_{map} involves only local computations and scales trivially. Scaling $T_{shuffle}$ to a large number of nodes is much more demanding as this requires linear scaling of the bisection bandwidth to the number of nodes—the network cost increases superlinearly to the number of nodes to scale the bisection bandwidth proportional to p. Disk I/O overhead—the representative MapReduce runtime systems store intermediate data in disks instead of DRAM—increases T_{map} and lowers the bisection bandwidth requirement. If $f \ll 1$, the bisection bandwidth requirement is even lower. If f is large, however, the bisection bandwidth is likely to become a bottleneck if the system size becomes very large.

Disk I/O overhead is unavoidable for workloads that overflow the aggregate DRAM capacity of a MapReduce cluster—*e.g.* to store the large input graph in our problem. If workloads' memory footprint fits into the aggregate DRAM capacity, however, the relatively low disk I/O bandwidth compared to the DRAM bandwidth incurs a significant performance overhead—*e.g.* finding a shortest path in the smaller subgraph.

5.2.3 Finding Shortest Paths in the Subgraph

To find shortest paths in the subgraph, we first need to extract the subgraph from the larger input graph (see Algorithm 5.1). For the subgraph extraction, $f \leq 0.01 \ll 1$ in our problem, and a single MapReduce iteration is sufficient as discussed in Section 5.2.1. W_{map} dominates the execution time as $f \ll 1$, and W_{map} is $O(m)$, where m is a number of edges in the input graph. The best sequential algorithm also requires the asymptotically same amount of work, and the MapReduce algorithm is optimal under our optimality criterion. The bisection bandwidth requirement is also low as the volume of communication in the shuffle phase—$O(mf)$—is much smaller than the amount of the disk I/O and computation in the map phase—$O(m)$. The disk I/O overhead in reading the input graph is unavoidable, and the amount of disk I/O in the following phases are much smaller.

```
Input: an edge list for the input graph
Output: adjacency lists for the subgraph
map(key/* unused */, edge/* connects head and tail */) {
  if (edge belongs to the subnetwork of interest)
    output (head, tail);/* a (key, value) pair */
    output (tail, head);
  }
}
reduce(vertex, adjacent_vertices) {
  degree = number of vertices in adjacent_vertices;
  output (vertex, degree + `` '' + adjacent_vertices);
}
```

Algorithm 5.1: Extracting a subgraph using MapReduce.

Once we extract the subgraph, the next step is finding a single-pair shortest path in the subgraph. This requires multiple MapReduce iterations (say k iterations) as analyzed in Section 5.2.1. Our shortest path algorithm extends the breadth-first search algorithm designed for the MapReduce programming model (see [20]) and sets initial distances from the two vertices in the pair (say s and t) first—0 for the source vertex and ∞ for the others. Then, our implementation invokes Algorithm 5.2 (*iteration_number* is passed as a command line argument) repeatedly to expand the breadth-first search frontiers from s and t by one hop in each invocation.

Assume m/n is constant, where n is the number of vertices and m is the number of edges. Then, setting initial distances from s and t requires only the map phase and costs $O(n)$ work. For Algorithm 5.2, $f \geq 1$, $fr \simeq 1$, and f varies only slightly during k iterations. Each invocation of Algorithm 5.2 costs $O(n + nf) + p \times Sort(\frac{nf}{p})$. The amount of work to find the shortest path becomes $(O(n + nf) + p \times Sort(\frac{nf}{p})) \times k$. The best sequential algorithm for breadth-first search runs in $O(n)$, and the work required to find the shortest path—by running breadth-first searches from both s and t with the heuristic of expanding the smaller frontier of the two—is even lower as the algorithm visits only a portion of the graph.

```
Input: adjacency lists for the subgraph with distances from
$s$ and $t$ ($distance\_s$ and $distance\_t$).
Output: adjacency lists for the subgraph with updated
distances from $s$ and $t$.
map($key$/* unused */, $adjacency\_list$) {
  parse $adjacency\_list$ to find $vertex$, $adjacenct\_vertices$,
  $distance\_s$ and $distance\_t$;
  remove $vertex$ from $adjacency\_list$;
  output ($vertex$, $adjacency\_list$);
  if($distance\_s$ = $iteration\_number$) {
    output ($neighbor$, 's') for every element $neighbor$ of
    $adjacenct\_vertices$;
  }
  if($distance\_t$ = $iteration\_number$) {
    output ($neighbor$, 't') for every element $neighbor$ of
    $adjacenct\_vertices$;
  }
}
reduce($vertex$, $values$) {
  $new\_distance\_s = \infty$;
  $new\_distance\_t = \infty$;
  while ($values$ is not empty) {
    $value$ = remove $values$' next element;
    if($value$ is 's') $new\_distance\_s = iteration\_number + 1$;
    else if($value$ is 't') $new\_distance\_t = iteration\_number$
    $+ 1$;
    else $adjacency\_list = value$;
  }
  parse $adjacency\_list$ to find $distance\_s$ and $distance\_t$;
  if($distance\_s < new\_distance\_s$) $new\_distance\_s = distance\_s$;
  if($distance\_t < new\_distance\_t$) $new\_distance\_t = distance\_t$;
  replace $distance\_s$ and $distance\_t$ in $adjacency\_list$ with
  $new\_distnace\_s$ and $new\_distance\_t$;
  output ($vertex$, $adjacency\_list$);
}
```

Algorithm 5.2: Expanding breadth-first search frontiers from s and t using MapReduce.

Retrieving the shortest path from the output is also problematic as the MapReduce programming model lacks a random access mechanism. In the RAM model, visiting at most k vertices and their neighbors in the backward directions towards s and t is sufficient to retrieve the shortest path. In the MapReduce programming model, in contrast, retrieving the shortest path requires additional MapReduce iterations and scanning the entire graph in each iteration. Thus, the MapReduce algorithm to find a single-pair shortest path is clearly suboptimal under our optimality criterion.

Bisection bandwidth is likely to become a bottleneck for large systems as the volume of communication is comparable or larger than the amount of work in the map phase, or $f \geq 1$. The overhead of the MapReduce runtime system and disk I/O exacerbates the situation as the workload incurs multiple MapReduce iterations with large volumes of the intermediate data. Cohen [37] presented several graph algorithms under the MapReduce programming model, and these algorithms also have large f and require multiple MapReduce iterations. Many of the algorithms require more work than the best sequential algorithm and necessitate a large bisection bandwidth to scale in large systems. The performance of these algorithms may not be good on large systems with limited bisection bandwidth as the author also commented.

5.3 A Highly Multithreaded System

Highly multithreaded systems support the shared memory programming model with the efficient latency hiding mechanism via multithreading and relatively large memory bandwidth and network bisection bandwidth for the system size. The shared memory programming model provides a randomly accessible global address space to programmers.

5.3.1 Algorithm Level Analysis

Highly multithreaded architectures adopt the shared memory programming model with a randomly accessible global address space, which matches well with irregular data access patterns over large data. A memory reference requires only a comparable number of instructions to the ideal RAM model. Memory access latency is higher than the ideal RAM model, but a large number of threads efficiently hide the latency assuming sufficient parallelism.

Bader *et al.* [14] provided a complexity model for highly multithreaded systems by extending Helman and JáJá's work [53] for symmetric multiprocessors (SMPs). This model encompasses the architectural characteristics of highly multithreaded systems. For SMPs, running time to solve a problem of size n with p processors, or $T(n,p)$ is expressed by the triplet $\langle T_M(n,p),\ T_C(n,p),\ B(n,p)\rangle$, where $T_M(n,p)$ is the maximum of the number of non-contiguous memory accesses by any processor, $T_C(n,p)$ is the maximum of local computational complexity of any processor, and $B(n,p)$ is the number of barrier synchronizations. This model penalizes non-contiguous memory accesses with higher latency and a large number of barrier synchronizations. Bader *et al.* [15] extended this model for multicore architectures as well. For highly multithreaded architectures, memory access latency is efficiently hidden, and a non-contiguous memory access costs $O(1)$. Multithreading also reduces $B(n,p)$. Bader *et al.* [14] concluded that considering only $T_C(n,p)$ is sufficient for highly multithreaded systems. An algorithm with $T_C(n,p)$ time complexity has at most $p \times T_C(n,p)$ work complexity.

5.3.2 System Level Analysis

Highly multithreaded systems are known to be efficient for applications with a large number of irregular data accesses [14, 16, 17, 77] and also lower programming burden to consider data locality. These are mainly due to relatively large memory bandwidth—for single-node systems—or bisection bandwidth—for multi-node systems—to the system size in addition to the efficient latency hiding mechanism.

Traditional microprocessors with powerful integer and floating point execution units—but only one thread per core—suffer from low processor utilization for latency-bound workloads. Highly multithreaded architectures, in contrast, achieve high processor utilization for those workloads [16]. Thus, highly multithreaded systems can perform a same amount of computation using a smaller number of processors than a system with the traditional microprocessors. A typical highly multithreaded system consists of a small number of tightly integrated nodes with large memory bandwidth and network bisection bandwidth, and this addresses the communication issue in bandwidth-bound applications with random access patterns as well. However, these systems have limited aggregate computing power, memory and disk capacity, and I/O bandwidth as a downside due to a small number of nodes in the system.

5.3.3 Finding Shortest Paths in the Subgraph

In order to extract the subgraph from the input graph of multiple terabytes, we first need to store the input graph in a highly multithreaded system. For a single node system with a small number of disks, even storing the input data is not possible, and our problem cannot be solved. If the input graph fits into the capacity, then we can find shortest paths in the subgraph as the following.

1. Read the graph data from the disks and filter the edges in parallel and in a streaming fashion, store only the filtered edges in the main memory.
2. Transform the edge list with the filtered edges to the adjacency lists.
3. Run the single-pair shortest path algorithm optimized for highly multithreaded systems and complex networks (see [16]).

For the first step, the aggregate disk I/O bandwidth or the network bandwidth to the file server—when the input graph is stored in the separate file server—limits the data read rate. As typical highly multithreaded systems have limited disk I/O bandwidth or the network bandwidth to the separate file server, reading the entire input graph data takes a significant amount of time. Transforming the edge list to the adjacency lists is straightforward with the random access capability. Bader and Madduri [16] designed an efficient algorithm to find the shortest path on the Cray MTA, and the SNAP package [18], which is portable to shared memory architectures, also provides a breadth-first search implementation that matches well with highly multithreaded architectures and complex networks. We tune this breadth-first search implementation to find a single-pair shortest path with a heuristic of expanding the smaller frontier (see [16]). This algorithm has the asymptotically same work complexity compared to that of the best sequential algorithm.

System	MapReduce cluster	highly multithreaded	hybrid
Nodes (# nodes, type)	(4, IBM System x3755)	(1, Sun SPARC T5120)	MapReduce cluster + highly multithreaded
Processors (# processors, type, power) / node	(4, AMD Opteron 2.4 GHz 8216, 95 W / processor)	(1, Sun UltraSparc T2 1.2 GHz, 91 W / processor)	
DRAM size / node	8 GB	32 GB	
DRAM bandwidth / node	42.8 GB/s	60+ GB/s	
Interconnect	Dual-link 1 Gb/s Ethernet	N/A	
Disk Capacity	96 disks × 1 TB / disk	2 disks × 146 GB / disk	
Software	Hadoop 0.19.2, Sun JDK 6	Sun Studio C compiler 5.9	

Table 7: Technical specifications for the test platforms

5.4 The Hybrid System

Analyzing a large scale complex network imposes distinct computational challenges (see Section 5.1), which cannot be efficiently served with a MapReduce cluster or a highly multithreaded system on its own (see Section 5.2 and 5.3).

We design a novel hybrid system to address the computational challenges in large scale complex network analysis. Our hybrid system tightly integrates a MapReduce cluster and a highly multithreaded system and exploits the strengths of both in a synergistic way. In each step of the complex network analysis, we select a MapReduce cluster or a highly multithreaded system—based on the match between the computational requirements and the architectural capabilities—and perform the computation for the step. We transfer the data from one system to the other if we switch from one to the other in the consecutive steps, and the tight integration of the two systems reduces the data transfer time.

5.4.1 Algorithm Level Analysis

Solving our problem, which finds single-pair shortest paths in the subgraph, requires filtering the large input graph; transforming the edge list with the filtered edges to the adjacency lists; and running the shortest path algorithm on the subgraph. The MapReduce algorithm in Section 5.2.3 finishes the first two steps in a single

MapReduce iteration, and this algorithm is optimal (under our optimality criterion for MapReduce algorithms) assuming $f \ll 1$. A highly multithreaded system, in contrast, suffers from limited disk capacity and the disk I/O bandwidth to finish the first step. For the last step, the MapReduce algorithm necessitates a significantly larger amount of work than the best sequential algorithm, and the disk I/O and runtime system overhead further exacerbates the performance. The extracted subgraph—in the sparse representation—has a higher chance to fit into the main memory of a highly multithreaded system, and if this is the case, a highly multithreaded system can find shortest paths in an efficient way. Using a MapReduce cluster for the first two steps and a highly multithreaded system for the last step exploits the strengths of both architectures, and we need to transfer the output of the second step from the MapReduce cluster to the highly multithreaded system in this case.

We generalize the above discussion to design a model that estimates the time complexity on the hybrid system. Assume the computation requires l steps. Then, execution time on the hybrid system is

$$T_{hybrid} = \Sigma_{i=1\ to\ l}\ min(T_{i,\ MapReduce} + \Delta, T_{i,\ hmt} + \Delta)$$

where $\Delta = \frac{n_i}{BW_{inter}} \times \delta(i-1,i)$. $T_{i,\ MapReduce}$ and $T_{i,\ hmt}$ are time complexities for the ith step on the MapReduce cluster and the highly multithreaded system, respectively. n_i is input data size for the ith step, and BW_{inter} is the bandwidth between the MapReduce cluster and the highly multithreaded system. $\delta(i-1,i)$ is 1 if selected platforms for the $(i-1)$th and ith steps are different and 0 otherwise. As the input data resides in the MapReduce cluster, $\delta(0,1)$ is 0 for the MapReduce cluster and 1 for the highly multithreaded system.

5.4.2 System Level Analysis

In the hybrid system, data transfer between the MapReduce cluster and the highly multithreaded system is necessary in addition to computation on each. The hybrid system becomes more effective if this data transfer time is minimized, and the data transfer time reduces as BW_{inter} increases. Minimizing the distance between the MapReduce cluster and the highly multithreaded system and tightly coupling the two is important to provide large BW_{inter} and to maximize the effectiveness of the hybrid system.

5.4.3 Finding Shortest Paths in the Subgraph

We find shortest paths in the subgraph using the hybrid system through the following steps.

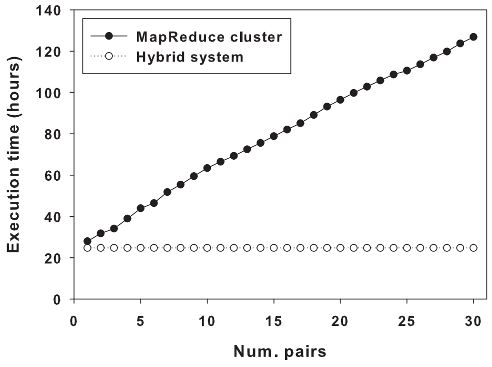
1. Extract the subgraph using the MapReduce cluster with the algorithms described in Section 5.2.3.
2. Switch to the highly multithreaded system, and load the extracted graph from the MapReduce cluster to the highly multithreaded system's main memory.
3. Find shortest paths using the highly mutltithreaded system by running the algorithm described in Section 5.3.3 multiple times.

As $f \ll 1$ in our problem, the size of the extracted graph is much smaller than the input graph. Thus, the subgraph has a higher chance to fit into the main memory of the highly multithreaded system. The MapReduce algorithm to solve the shortest path problem incurs significantly more work than the best sequential algorithm while the algorithm for the highly multithreaded system incurs only a comparable amount of work to the best sequential algorithm. For graphs that fit into the main memory, the disk I/O and runtime system overhead even widens performance gap between the MapReduce cluster and the highly multithreaded system. If the difference between execution time on the MapReduce cluster and the highly multithreaded system to find single-pair shortest paths is larger than the subgraph loading time, using the highly multithreaded system for the third step reduces the execution time to solve our problem.

5.5 Experimental Results

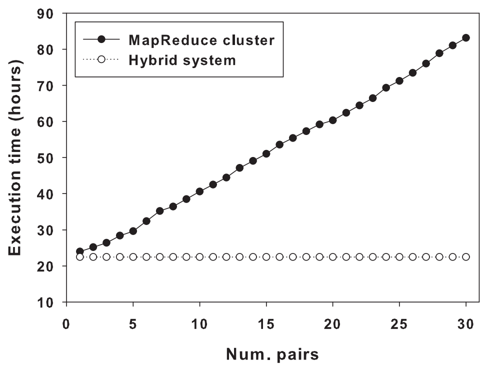
Table 7 summarizes the test platforms. Hadoop is configured to spawn up to 8 mapper processes and 3 reducer processes per node. Hadoop Distributed File System (HDFS) is configured to create one replica per block. We add an additional system with the Intel Xeon processors to coordinate the map and reduce processes for the MapReduce cluster.

Figures 37, 38, and 39 summarize experimental results in finding shortest paths in the subgraphs which cover 10%, 5%, and 2% of the vertices in the input graph, respectively. The extracted subgraphs have 351 million vertices and 2.75 billion undirected edges, 178 million vertices and 1.49 billion undirected edges, and 44.9 million vertices and 109 million undirected edges (excluding isolated vertices), respectively. We find single-pair shortest paths in the subgraphs for up to 30 pairs.



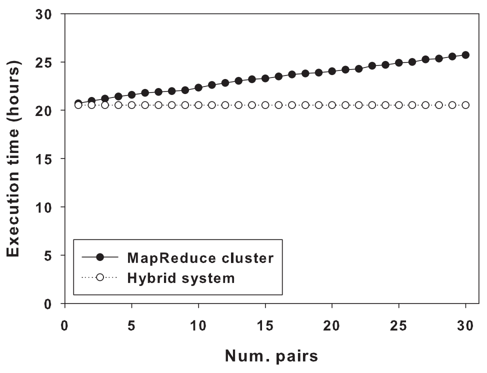
	MapReduce cluster (hours)	hybrid (hours)
subgraph extraction	23.9	23.9
memory loading	-	0.832
shortest paths (for 30 pairs)	103	0.000727

Figure 37: Execution time to extract the subgraph (which covers 10% of the vertices) and find single-pair shortest paths (left) and the decompositions of the execution time on the MapReduce cluster and the hybrid system (right). The highly multithreaded system fails to solve the problem on its own as the input graph overflows its disk capacity.



	MapReduce cluster (hours)	hybrid (hours)
subgraph extraction	22.0	22.0
memory loading	-	0.418
shortest paths (for 30 pairs)	61.1	0.000467

Figure 38: Execution time to extract the subgraph (which covers 5% of the vertices) and find single-pair shortest paths (left) and the decompositions of the execution time on the MapReduce cluster and the hybrid system (right). The highly multithreaded system fails to solve the problem on its own as the input graph overflows its disk capacity.



	MapReduce cluster (hours)	hybrid (hours)
subgraph extraction	20.5	20.5
memory loading	-	0.0381
shortest paths (for 30 pairs)	5.22	0.000191

Figure 39: Execution time to extract the subgraph (which covers 2% of the vertices) and find single-pair shortest paths (left) and the decompositions of the execution time on the MapReduce cluster and the hybrid system (right). The highly multithreaded system fails to solve the problem on its own as the input graph overflows its disk capacity.

The MapReduce cluster successfully extract the subgraphs, while the UltraSparc T2 blade fail to solve the problems on its own as the input graph size overflows the system's disk capacity. Using a network file system with a larger disk capacity is another option for the highly multithreaded system, but even in this case, the network bandwidth to the file system or the limited computing power of the highly multithreaded system is likely to become a performance bottleneck. Pre-processing the data before the transfer using a file system with a filtering capability—as is the case of the hybrid system—provides a better mechanism to work around these limitations. Finding single-pair shortest paths in the extracted subgraphs is significantly slower on the MapReduce cluster than the shared memory system. The hybrid system outperforms the MapReduce cluster in overall throughout the experiments. The performance gap widens as the subgraph size or the number of input pairs to find a shortest path increases. Adopting a faster interconnection technology between the MapReduce cluster and the highly multithreaded system will further widen the gap. This justifies the use of the hybrid system, especially when the analysis of the extracted subgraph requires a large amount of work.

The MapReduce cluster runs significantly slower than the highly multithreaded system in finding shortest paths. *This performance gap is mainly due to the match between computational requirements of the problem and the programming model and the architectural capability of the two different systems.* The MapReduce algorithm for the problem is clearly suboptimal under our optimality criterion while the algorithm for the highly multithreaded system executes only a comparable number of operations to the best sequential algorithm. The input pairs to find shortest paths are 6.04 hops away from each other in average (in the case of the 10% subnetwork, excluding the disconnected pairs) and the UltraSparc T2 blade visits approximately 100 thousand vertices in average—owing to the random access mechanism—with the heuristic of expanding the smaller frontier. The MapReduce algorithm needs to visit the entire vertices multiple times. The tightly connected network for the highly multithreaded system and the highly multithreaded system's latency hiding mechanism naturally match with the workload's irregular data access patterns, while the disk I/O and the runtime system overhead of the MapReduce cluster becomes more prominent with the graph data that fits within the main memory capacity.

5.6 Summary

Complex networks capture interactions among entities in various application areas in a graph representation. Analyzing large scale complex networks often answers important questions—*e.g.* estimate the spread of epidemic diseases—but also imposes computing challenges mainly due to large volumes of data and the irregular structure of the graphs.

In this chapter, we aim to solve such a challenge: finding relationships in a subgraph extracted from large data. We solve this problem using three different platforms: a MapReduce cluster, a highly multithreaded system, and a hybrid system of the two. The MapReduce cluster and the highly multithreaded system reveal limitations in efficiently solving this problem, whereas the hybrid system exploits the strengths of the two in a synergistic way and solves the problem at hand. In particular, once the subgraph is extracted and loaded into memory, the hybrid system analyzes the subgraph five orders of magnitude faster than the MapReduce cluster.

CHAPTER VI

PHYLOGENETIC TREE RECONSTRUCTION USING GENE ORDER DATA AND PARALLELIZING THE COGNAC SOFTWARE PACKAGE

This chapter discusses the parallelization of *COGNAC* software package which reconstructs a phylogenetic tree using gene order data. *We are under preparation to submit a paper based on the materials in this chapter.*

Gene order data captures the chromosome structure of a species. Phylogenetic trees reconstructed using gene order data are often more accurate than trees reconstructed using nucleotide sequence data for distant genomes, and gene order data based phylogenetic tree reconstruction methods are becoming popular. However, gene order data based reconstruction methods are computationally expensive, and parallel processing—along with the development of novel algorithms—is needed to address the computational challenges.

This chapter presents the parallelization of the *COGNAC* software package which reconstructs a phylogenetic tree of multiple species using their gene order data. *COGNAC* has parallelisms in multiple levels. All those combined, *COGNAC* provides a very high degree of parallelism. However, parallelization points in *COGNAC* are heavily nested, and the degree of parallelism in each parallelization point varies widely based on the input data and also throughout the computing phases. Exploiting heavily nested and irregular parallelism is very challenging without a proper system software support, and there are several attempts to solve the challenge by developing new libraries, languages, or compilers. We use Intel TBB and the Intel Nehalem architecture with simultaneous multithreading support to parallelize the *COGNAC* software

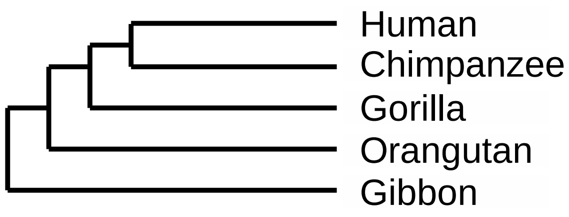


Figure 40: An exemplar phylogenetic tree [95].

package and achieve high scalability with little effort. This shows the importance of mapping an application to an appropriate programming model and an architecture. The remainder of this chapter explains gene order data based phylogenetic tree reconstruction and our approaches to solve the problem.

6.1 Phylogenetic Tree Reconstruction Using Gene Order Data

A phylogenetic tree (see Figure 40) captures speciation events among multiple organisms. Constructing a phylogenetic tree requires inferring ancestral relationship among multiple organisms based on currently available data. Traditionally, scientists had studied this problem by inspecting fossils or comparing the morphology and the physiology of living creatures, but these approaches revealed limitations due to an incomplete set of fossils or the complex nature of evolutionary mechanisms affecting the morphology and the physiology of organisms [95].

The increasing availability of genetic data (Table 8 explains genetic data in different levels) opens a new opportunity to solve the problem. Constructing a phylogenetic tree by comparing nucleotide sequences of a single gene or a few genes has been intensively studied, and there are several representative approaches. The neighbor-joining method [111] is a heuristic, and the method greedily merges a pair of genomes based on the minimum evolution principle. Maximum parsimony (MP) methods (*e.g.* [47, 52]) find a topology with the minimum number of mutations (or the lowest parsimony score), and maximum likelihood (ML) methods (*e.g.* [28, 45])

Table 8: Basic units of genome sequence data in different levels (genome $\ni$ chromosome $\ni$ gene $\ni$ nucleotide).

genome	the genome of an organism captures all the genetic information for the organism.
chromosome	a chromosome is a thread-like structure in a cell, and a chromosome contains multiple genes.
gene	a gene is a basic unit that affects a trait of an organism and consists of multiple nucleotides.
nucleotide	a nucleotide is a basic building block of DNA and RNA. For example, DNA consists of four types of nucleotides: adenine (A), thymine (T), cytosine (C), and guanine (G). Uracil (U) replaces thymine (T) for RNA.

attempt to find a tree with the highest likelihood value under a certain evolutionary model. MP and ML methods are generally more accurate than the neighbor-joining method but also more computationally expensive. These methods, using nucleotide sequence data, find a reasonably accurate tree topology for close genomes. For distant genomes, these methods become significantly less accurate due to the high rate of nucleotide sequence level mutations.

Sankoff and Blanchette [112] pioneered in inferring a phylogenetic tree using gene order data [112], and the following explains gene order data. The genome of an organism captures all the genetic information for the organism, and a nuclear genome of many species consists of multiple chromosomes. There are multiple genes in a chromosome. We can also identify a set of genes originated from a common ancestral gene. If two genes (possibly located in chromosomes of different species) are originated from a common ancestral gene, scientists say the two genes are homologous. If we assign a unique number to a set of homologous genes, we can represent a chromosome as a sequence of numbers; numbers appear in the order the genes corresponding to

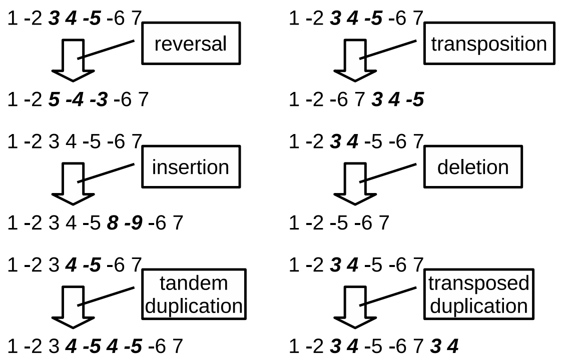


Figure 41: Intra-chromosomal genome rearrangement events. Each gene is represented by an unsigned number and its strandedness (in the double stranded structure of DNA, some genes are found in one strand and read in a direction for the strand while other genes are found in another strand [113].) is represented by the sign of the number.

the numbers appear in a chromosome. If a genome has multiple chromosomes, we can represent the genome with multiple sequences of numbers—one sequence per chromosome. Such sequences of numbers are gene order data.

Genome rearrangement events (see Figures 41 and 42 for examples) change the structure of chromosomes (or reorder numbers in gene order data). By comparing gene order data of multiple organisms, we can infer mutations occurred in chromosome level. Chromosome level mutations are significantly less frequent than nucleotide sequence level mutations and have a higher impact in speciation. This enables us to reconstruct a more accurate tree for distant genomes.

ML methods require a good model to simulate the evolutionary process, but we do not have such a model for gene order data yet. Therefore, MP methods are widely used for gene order data. MP methods for gene order data require significantly more computing than MP methods for nucleotide sequence data. MP methods first enumerate candidate tree topologies; compute parsimony scores for the enumerated trees; and select the trees with the lowest parsimony score. For N organisms there are $(2N-5)!! = 3\times 5\times\cdots\times(2N-7)\times(2N-5)$ possible unrooted candidate tree topologies, this makes MP methods challenging for large N. Scoring a single tree topology costs

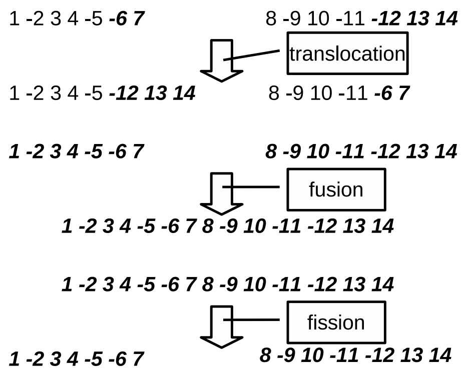


Figure 42: Inter-chromosomal genome rearrangement events. Each gene and the gene's strandedness are represented by an unsigned number and its sign, respectively. Two sequences represent gene order data of two chromosomes.

only a polynomial number of operations [95] for nucleotide sequence data. For gene order data, scoring a single topology is NP-hard. There is no known algorithm to find the most parsimonious labeling of the internal genomes in a tree with more than three leaf genomes [91]. Even heuristics [46] to score a single topology require solving many NP-hard median problems. A median genome of three genomes is the genome that minimizes the sum of distances between the genome and the three genomes. A median problem finds a median genome. MP methods reconstruct internal genomes and score a tree topology by solving multiple median problems till the tree score converges. Yet, a median problem is NP-hard even with linear-time computable pairwise distance metrics (*e.g.* breakpoint distance, inversion distance, and DCJ distance). Actually, computing the distance between two genomes can be NP-hard based on the definition of the distance [46]. All these combined, constructing a phylogenetic tree using gene order data necessitates an enormous amount of computing.

6.2 Disk-covering methods and GRAPPA

Warnow and her group [56, 57, 107] proposed several disk-covering methods (DCMs) to reduce candidate tree search space—recall that we need to consider $(2N-5)!!$ candidate tree topologies for N species with a brute force method. DCMs decompose the input genomes to multiple overlapping disks; find a tree topology for each disk; and merge the topologies to reconstruct a tree for the entire input genomes. Rec-I-DCM3 [107], which is the most recently published DCM, recursively decomposes the input genomes to further reduce the search space and iterates the process multiple times to refine the reconstructed tree. Roshan *et al.* reported that Rec-I-DCM3 reconstructed a highly accurate tree for very large N (up to 13,921). However, existing DCMs have several shortcomings. Computing a disk decomposition is expensive for the original DCM [56] and DCM2 [57]. Rec-I-DCM3 computes a decomposition faster but requires multiple iterations to achieve high accuracy. The existing DCMs place a significant number of genomes in the overlapping region, and this also increases computing time.

GRAPPA (Genome Rearrangements Analysis under Parsimony and other Phylogenetic Algorithms) [91, 87, 89, 90, 118, 119]—along with MGR [26]—is the most accurate software package for gene order data based phylogenetic tree reconstruction. The first version of GRAPPA is based on BPAnalysis [112], and the GRAPPA was implemented using several high-performance computing techniques to accelerate BP-Analysis. GRAPPA has been updated multiple times, and DCM-GRAPPA combines GRAPPA and DCM2.

6.3 COGNAC

We designed a new DCM and the *COGNAC* (**C**omparing **O**rders of **G**enes using **N**ovel **A**lgorithms and high-performance **C**omputers) software package [67]. The new DCM remedies the shortcoming of existing DCMs, and *COGNAC* marries GRAPPA with the new DCM. The new DCM is based on the spectral method. The DCM recursively decomposes the input leaf genomes to two smaller and possibly overlapping sets—or disks. The method first constructs a Laplacian matrix using the pairwise distances between the leaf genomes in a disk and uses an eigenvector for the second smallest eigenvalue of the Laplacian matrix to find an initial bi-partitioning of the leaf genomes. The method applies a heuristic to refine the initial bi-partitioning, and the heuristic places several genomes in both disks in certain cases. The new DCM recursively decomposes a disk till three or less genomes are left and builds a binary disk tree (see Figure 43).

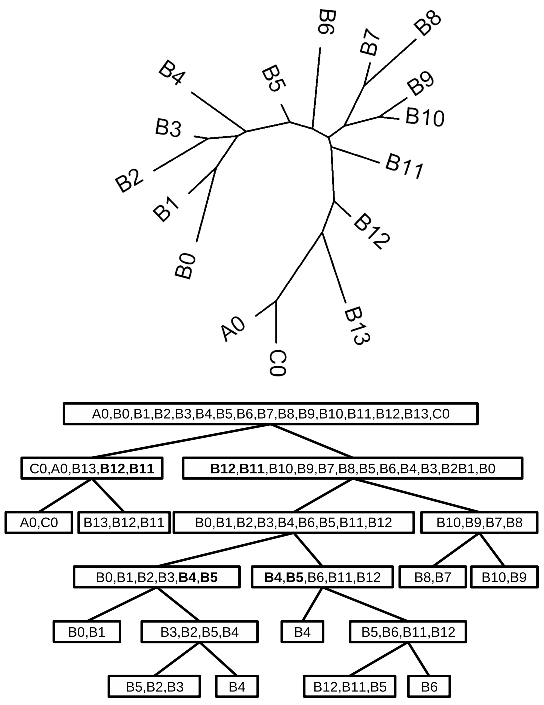

B0
B1
B2
B3
B4
B5
B6
B7
B8
B9
B10
B11
B12
B13
A0
C0
A0,B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,C0
C0,A0,B13,**B12**,**B11**
B12,**B11**,B10,B9,B7,B8,B5,B6,B4,B3,B2B1,B0
A0,C0
B13,B12,B11
B0,B1,B2,B3,B4,B6,B5,B11,B12
B10,B9,B7,B8
B0,B1,B2,B3,**B4**,**B5**
B4,**B5**,B6,B11,B12
B8,B7
B10,B9
B0,B1
B3,B2,B5,B4
B4
B5,B6,B11,B12
B5,B2,B3
B4
B12,B11,B5
B6

Figure 43: A model phylogenetic tree and a disk tree for the tree. The disk tree is constructed using only the pairwise distances between the leaf genomes in the model tree. Genomes placed in the overlapping region of two disks are marked in bold.

To reconstruct a tree for the entire input genomes, *COGNAC* traverses the binary disk tree in a bottom up fashion and merges the reconstructed trees of two child disks to reconstruct a tree for their parent disk. In merging two disks, *COGNAC* enumerates multiple candidate trees first and selects the trees with the lowest parsimony score. Algorithm 4 summarizes *COGNAC*'s phylogenetic tree construction algorithm.

```
Input: Gene order data for the input genomes.
Output: An output phylogenetic tree of the input genomes.
Compute the $\frac{(n-1)\times(n-1)}{2}$ pairwise distances between the $n$ input genomes.
Apply the DCM in a recursive way and construct a binary disk tree.
Push all the leaf disks to a work queue.
while the work queue is not empty do
    Pull out a disk from the work queue.
    if the disk is a leaf disk then
        if the disk has three genomes then
            Build a tree topology using the three leaf genomes.
            Solve a median problem to initialize the internal genome in the tree.
        end
    else
        Enumerate candidate tree topologies by merging the disk's two child disks.
        Score the enumerated candidate tree topologies.
        Select the trees with the lowest parsimony score.
    end
    If the processed disk's sibling disk is also processed, push the disk's parent
    disk to the work queue.
end
```

Algorithm 4: A high-level overview of *COGNAC*'s phylogenetic tree construction algorithm.

6.4 Nested Irregular Parallelism in COGNAC

GRAPPA—without DCM—needs to score a very large number of trees. Scoring a single tree is compute-intensive as well. Moret *et al.* [87] had parallelized GRAPPA by scoring different candidate trees using different processors and achieved linear speedup up to 512 processors. However, with the new DCM, *COGNAC* needs to score only a small number of tree topologies per disk; we cannot achieve such a speedup by simply scoring different trees using different processors (or cores). Yet, we can still achieve a high level of scalability by using parallelism in multiple levels. For example, we can process all the leaf disks in a binary disk tree in parallel. Also, in scoring a single topology, we can solve multiple median problems in parallel. Table 9 presents a part of the parallelism in *COGNAC*.

Table 9: A part of the parallelism in *COGNAC*.

parallelism	**comment**
We can compute the pairwise distances between the input genomes in parallel.	Computing the pairwise distances has a $O(N^2)$-way parallelism.
We can compute the distance between two genomes using multiple cores.	-
We can process multiple leaf disks in a disk tree in parallel.	Accessing the work queue holding disks—which are ready to be processed—requires data synchronization.
We can score multiple tree topologies for a single disk in parallel.	We can apply the branch-and-bounding approach. If a lower-bound of a tree score is larger than the current best tree score, we do not need to score the tree. This requires coordination among the threads scoring tree topologies for a single disk.
We can solve multiple median problems in parallel to score a single tree topology.	We can remove the data synchronization issue by an algorithm modification.
We can parallelize a median solver.	We can mix breadth-first search and depth-first search based approaches. The breadth-first based approach provides a higher level of parallelism but is less efficient and requires more memory. The depth-first search based approach is sequential but more efficient and has a small memory footprint.

COGNAC has easily identifiable parallelisms in multiple levels. Exploiting parallelisms often involves data synchronization issues, but identifying critical sections is mostly straightforward. For example, to process multiple disks in parallel, we need to protect the work queue using data synchronization primitives. One exception is the following. We can update multiple internal genomes in parallel to score a single tree topology, but this involves a non-trivial data synchronization issue. To score a tree, *COGNAC* first reconstructs the internal genomes and computes and sums the lengths of the edges in the tree. To reconstruct the internal genomes, *COGNAC* solves one median problem per one internal genome and initializes an internal genome to a median genome of the internal genome's three closest leaf genomes. Then, *COGNAC* updates internal genomes by replacing an internal genome with a median genome of the internal genome's three neighboring genomes till the tree score converges. Yet, if one of the three neighboring genomes is updated by another thread while computing a median genome, this can lead to an unpredictable result. Figure 44 illustrates the point.

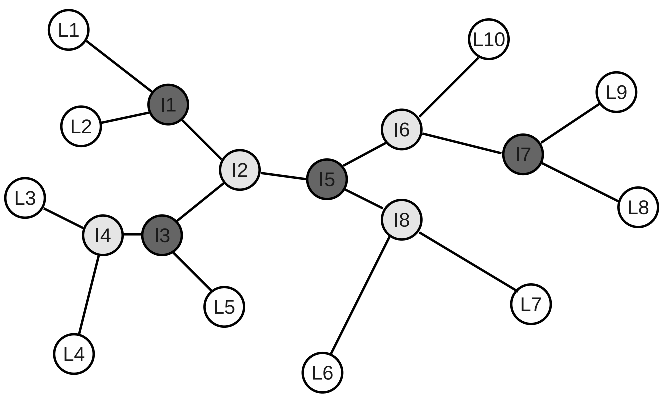


Figure 44: An illustration of the data synchronization issue in scoring a single tree topology using multiple threads.

To update I2 in the figure, *COGNAC* needs to find a median genome of I1, I3, and I5. Another thread can update I5 while solving the median problem. This raises a data synchronization issue. We can assign a lock to every internal genome to avoid the problem, but this significantly complicates the implementation. Instead, *COGNAC* divides the internal genomes to two groups. One group includes all the internal genomes which are an odd number of hops away from one leaf genome, and another group includes all the internal genomes which are an even number of hops away from the leaf genome. In the figure, all the internal genomes in dark-gray falls into one group, and the remaining internal genomes (in light-gray) belongs to another group. Then, *COGNAC* updates two groups in turn—but updates all the internal genomes in one group in parallel. This solves the data race problem at the cost of halving the "best" case parallelism.

All the parallelism combined, *COGNAC* has a very high degree of parallelism. In high-level, we can scale *COGNAC* to a large number of cores by exploiting all the parallelism in *COGNAC* by creating an "appropriate" number of threads in every parallelization points. If one thread has no work but there is a thread with too much work, we can steal work from the thread with too much work to achieve load balancing. However, transforming this high level algorithm to low level code can be very difficult if we do not use an appropriate language, compiler, or library.

6.5 Parallelizing COGNAC with Intel TBB

There are several programming languages, compilers, and Libraries that support nested parallelism. Blumofe *et al.* [21] implemented Cilk, and Cilk supports nested parallelism and work stealing. More recently, Intel announced Thread Building Block (TBB) [106]. DARPA had launched the High Productivity Computer Systems (HPCS) project, and several companies released experimental languages [76] to support high productivity in parallel programming—*e.g.* IBM's X10, Cray's Chapel, and Sun's Fortress. Intel TBB targets shared memory systems, and HPCS languages are mainly for large scale supercomputers. Intel TBB adopts a library based approach. X10, Chapel, and Fortress are newly designed languages.

We parallelize *COGNAC* using Intel TBB. We implemented *COGNAC* considering parallelization from the very beginning, and potential parallelization points and critical sections are already marked in the code. The parallelization work is mechanical with one exception; we use *parallel_do* in one place (to process the work queue for a disk tree), and we restructure the code to use *parallel_do*. *COGNAC* has 116 marked parallelization points, and we enable 30 of the 116 parallelization points using TBB. We guard critical sections using two different mechanisms—locks and OpenMP critical sections. Code 6.1 and 6.2 present an original *COGNAC* code segment and the modified segment to use Intel TBB, respectively. OpenMP currently guards all the critical sections using a single lock, but Transactional Memory can replace the current implementation in the future.

```
//PARALLEL
for(int i = 0 ; i < v_tree0.size() ; i++) {
 //PARALLEL
 for(int j = 0 ; j < v_tree1.size() ; j++) {
  vector<PhylTree> v_tmpEnumTree;
  enumCandidateTreesFromTwoOvlpTrees(v_leafGnm, vv_dist, v_exclGnmIdx0,
  v_ovlpGnmIdx, v_tree0[i], v_exclGnmIdx1, v_tree1[j], v_tmpEnumTree);
  //BEGIN_ATOMIC
  v_enumTree.insert(v_enumTree.end(), v_tmpEnumTree.begin(),
  v_tmpEnumTree.end());
  //END_ATOMIC
 }
}
```

Code 6.1: A code segment from the original *COGNAC*

```
parallel_for(blocked_range<int>(0, v_tree0.size()), [&](blocked_range<int>& r) {
for(int i = r.begin() ; i < r.end() ; i++) {
 parallel_for(blocked_range<int>(0, v_tree1.size()), [&](blocked_range<int>& r2) {
 for(int j = r2.begin() ; j < r2.end() ; j++) {
  vector<PhylTree> v_tmpEnumTree;
  enumCandidateTreesFromTwoOvlpTrees(v_leafGnm, vv_dist, v_exclGnmIdx0,
  v_ovlpGnmIdx, v_tree0[i], v_exclGnmIdx1, v_tree1[j], v_tmpEnumTree);
#ifdef __LOCK
  omp_set_lock(&lock);
#else
  #pragma omp critical
  {
#endif
  v_enumTree.insert(v_enumTree.end(), v_tmpEnumTree.begin(),
  v_tmpEnumTree.end());
#ifdef __LOCK
  omp_unset_lock(&lock);
#else
  }
#endif
 }
 } );
}
} );
```

Code 6.2: the modified *COGNAC* code segment to use Intel TBB

6.6 Experimental Results

We experiment using a system with two Nehalem-EP (E5530 2.4 GHz) processors. A single Nehalem-EP processor has four cores and two hardware threads per core. We use a random tree described in [119] as input data. The generator in [119] accepts an average edge length (say r) and the deviation of edge lengths (say d) as input parameters. We set r=8 and d=8. The generator sets edge lengths by randomly sampling an integer value between r - d and r + d. Figure 45 presents the results, and Table 10 shows several statistics collected using Intel Vtune.

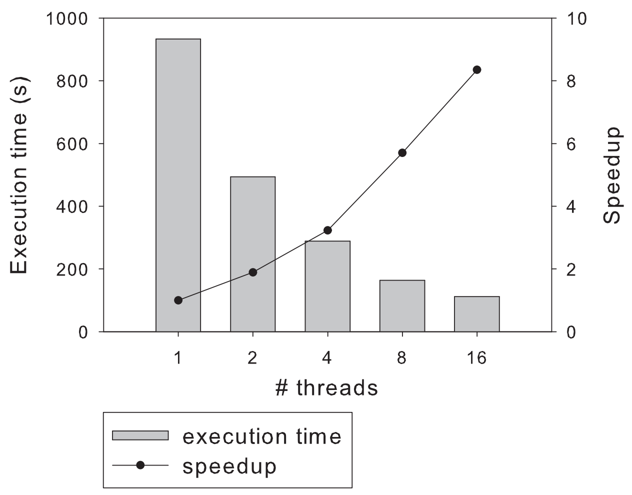


Figure 45: Execution time and speedup on a 8-core 16-threads machine (two Nehalem-EP processors).

Table 10: Performance statistics collected using Intel Vtune.

# threads	**# retired instructions**	**CPI**	**# local DRAM accesses**	**# remote cache & DRAM accesses**
1	4,210,411,175,936	0.59	62,300,000	6,400,000
8	4,267,145,691,136	0.66	48,200,000	84,300,000
16	4,452,142,546,944	0.89	52,800,000	91,600,000

We observe 5.7 times speedup with 8 cores and 8.4 times speedup with 16 threads (8 cores × 2-way simultaneous multithreading). 5.7 times speedup is sub-optimal. There are several factors for the result. First, the Intel Nehalem processor supports turbo boosting [30]. If a chip is under low utilization (*e.g.* if we are using just one core), the Nehalem processor increases its clock up to two frequency steps (one frequency step is 133 MHz). We cannot achieve 8 times speedup if the chip runs in lower clock speed with eight threads. The number of executed instructions also increase with eight threads; our system executed 1.3% more instructions with eight threads than a single thread case. TBB overhead is one source of the increase. *COGNAC* can score more trees when it runs in parallel, and this is the second source of the increase. Before *COGNAC* scores a tree, *COGNAC* checks whether the tree's lower-bound is lower than the current best score or not. If the lower-bound is higher than the current best score, *COGNAC* does not score the tree. If multiple threads score multiple trees in parallel, a thread can start scoring a tree with the high lower-bound before another thread finishes scoring a tree and updates the current best score to a lower value. The increase in a number of off-chip data transfers is the most significant factor for the suboptimal speedup. If you spawn a larger number of threads, those threads need to share L3 cache. This reduces the effective cache size per thread unless there is a high degree of data sharing among the threads. Also, if a thread running on one chip spawns its child thread on another chip, the child thread needs to read data from the remote L3 cache or remote DRAM. We observe 93% and 13 times increases in the number of off-chip data accesses and remote cache and DRAM accesses with 8 threads (than a single-threaded case), respectively. The CPU utilization is mostly close to 800% but is lower than 800% at the beginning and the end of the computations. This also contributes to the suboptimal speedup.

Simultaneous multithreading largely compensates the slowdown due to the increase in off-chip data accesses, and *COGNAC* achieved 8.3 times speedup. CPI per thread is 0.89. This translates to per core CPI of 0.45, which is higher than the single-threaded case. Each core in the Nehalem processor can issue up to four instructions per cycle, and this leads to the ideal CPI of 0.25. With simultaneous multithreading, *COGNAC* achieves a reasonable fraction of the ideal CPI. This results show the effectiveness of Intel TBB and simultaneous multithreading for nested irregular applications. Using the two, we achieve high scalability with little additional work.

6.7 Summary

In this chapter, we study the parallelization of our *COGNAC* software package which reconstructs a phylogenetic tree using gene order data. Using gene order data enables scientists to infer a phylogenetic tree for distant genomes. However, reconstructing a phylogenetic tree using gene order data is much more expensive than traditional approaches based on nucleotide sequence data. *COGNAC* adopts our newly designed disk covering method to reduce the number of tree topologies to be inspected. This is effective in reducing the amount of computation but also reduces the degree of parallelism. GRAPPA achieved high scalability by scoring different trees in parallel, but *COGNAC* cannot achieve high scalability by simply scoring multiple tree topologies in parallel. We exploit parallelism in multiple levels to provide sufficient parallelism and use Intel TBB to reduce the programming complexity in managing nested irregular parallelism. Using the combination of Intel TBB and simultaneous multithreading, we achieve 8.3 times speedup on a system with 8 cores with little additional effort. The result show that the complexity in parallel programming can be significantly lowered by using proper system software and with architectural support.

CHAPTER VII

CONCLUSIONS

Multi-core processors and many-core accelerators deliver higher performance per unit power consumption, but the superior power efficiency comes at the cost of higher software development complexity. There are several reasons for the increase in programming complexity. Some of those are unavoidable, but software tools and architectural support still can substantially improve software productivity. Researchers already have proposed multiple ideas, and the remaining question is identifying a right set of tools and architectural primitives as developing new software tools or architectural features involves additional cost. To answer the question, we need to understand the future application requirements and also need to test such ideas using appropriate benchmarks. We present the parallelization and the performance tuning of various kernels and applications with widely varying computational characteristics for diverse architectures in this dissertation. This chapter discusses major challenges in software development for modern parallel computers based on our experiences and presents directions to improve software productivity from an application- and algorithm-centric viewpoint.

7.1 Major Challenges in software development for Modern Parallel Computers

Parallel programming is difficult for several reasons. First, programmers need to identify parallelism in their problem. This is not always easy, and many researchers have published papers by designing parallel algorithms for seemingly sequential problems [?, 61]. We expect designing parallel algorithms for such problems will remain as a challenging research topic. However, many computationally expensive problems have easily identifiable parallelism—though there is a large gap between identifying parallelism in high level and implementing efficient and scalable software. For example, in JPEG2000, different code blocks can be encoded and decoded in parallel (in the EBCOT stage), or different lines can be processed in parallel (in the DWT stage). In reconstructing a phylogenetic tree, *COGNAC* can process different leaf disks in parallel; score different topologies in parallel; and solve multiple median problems in parallel. For a majority of computationally challenging workloads, identifying parallelism is not very difficult.

Scheduling and mapping are major sources of the gap between the identification of parallelism and the implementation of efficient and scalable code. Assigning a set of computations to appropriate computing units in proper timings is cruical to achieve load balancing and maximize the locality of computing. Without system software support, scheduling and mapping require a significant amount of coding effort especially for applications with nested and irregular parallelism—such as the phylogenetic tree reconstruction problem in this dissertation.

Data synchronization is another issue. Data synchronization involves two issues: identifying critical sections and guarding critical sections using proper data synchronization mechanisms. The first problem involves finding shared data to be protected and properly marking the start and the end of critical sections. These are not very different from the challenges SQL programmers are facing. However, the situation is much more challenging for programmers in the parallel computing domain. Using database transactions is mostly not an option due to their high performance overhead. There are a variety of mechanisms for data synchronization. There are different types of locks. Some locks use a busy-waiting mechanism while other locks use a queue based approach. Some locks allow multiple readers while other locks allow only a single reader or writer. Some locks support nesting while other locks do not. Multiple architectures support a different set of instructions to atomically update a single variable with low performance overhead. Programmers can consider lock free algorithms as well. Recently, Transactional Memory (TM) becomes a popular research topic, and there are several open-source (software) TM implementations. Programmers are asked to select different approaches for different cases, and this adds an additional burden. Lock based approaches are popular, but those approaches involve deadlock and livelock issues—especially with fine-grained locking. To avoid the issues, programmers often use only a small number of locks to guard all the critical sections in their code, but this limits scalability. This again forces them to spend a significant amount of time to minimize the size of critical sections. Atomic instructions work for only limited cases, and software TM incurs too high performance overhead to be a general solution. Lock free algorithms are hard to design and verify even for highly skilled programmers.

Nondetermistic execution is also a problem. Parallel computers interleave instructions in different orders in different runs. This complicates debugging and code verification. There are several ideas to support deterministic execution on parallel computers (*e.g.* [73, 99, 24]), but such approaches incur additional performance overhead, limit scalability, and also have not yet been reached their full maturity. For debugging, nondetermism is problematic mainly because it is difficult to reproduce a bug. However, if only debugging is a concern, we can afford relatively high performance overhead. Multiple research papers are available in this direction (such as [35, 85]). Nondeterminism is more problematic in writing correct code and verifying the code. There are too many possible interleaving scenarios, and interleaving scenarios in small granularities are very difficult to reason about. Yet, not all nondeterministic execution scenarios are equally bad. Also, we cannot completely remove nondeterminism—even in sequential computing as input data is nondeterministic. If we can limit the granularity of interleavings to the level that matches our high level intuition, the problem can be significantly alleviated. Our MSF algorithm using TM illustrates the point. Without TM, we need to reason about instruction level interleavings, and this is very challenging even for expert programmers. If we place a high level graph operation inside a transaction, we need to consider only interleavings that match our high level intuition. This largely reduces the possible interleaving scenarios and lowers programming and verification complexities.

Communication between a large number of computing units is also problematic. For single core systems, data access latency varies only moderately. Single core systems also provide relatively high memory bandwidth in comparison to their number crunching capability. For parallel computers, communication (bisection) bandwidth does not scale as fast as the core count. Data access latency also varies more widely than single core systems. GPU accelerators further increase programming complexity. GPUs have an impressive number crunching capability, but data need to be off-loaded to a GPU via a slower PCI-Express bus first to exploit the GPU (recall Chapter 2). Programmers need to maximize the temporal locality and the spatial locality of their computation. The loop merging algorithm for Discrete Wavelet transform in Chapter 4 improves temporal locality, but this type of code modification is far from trivial. Programmers also need to use block data transfers based on the analysis of data access patterns to achieve high network bandwidth utilization and hide data access latency. The Cell Broadband Engine delivers high bandwidth utilization owing to its block data transfer mechanism (Chapter 2), but the processor also suffers from low software development productivity. Programmers need to fine-tune data partitioning to minimize global communication especially for supercomputers and applications dealing with large matrices and vectors. Minimizing global communication often becomes the most difficult challenge in many supercomputing applications.

Addressing the data transfer issues for large scale supercomputers and communication-intensive applications may remain as a job for highly skilled programmers. Yet, emerging technologies—such as embedded DRAM cache (adopted in the IBM Power 7), 3D stacked DRAM, and CPU-GPU fusion chips—have strong potential to largely solve the problem at least for systems with a single chip multiprocessor. With the combination of embedded DRAM cache and hardware multithreading and for applications with a moderate memory footprint, data access latency or memory bandwidth is not an issue. For applications with a larger memory footprint, emerging 3D stacked DRAM can be a solution. AMD is expected to release CPU-GPU fusion chips in the near future. Intel and NVIDIA are also expected to release fusion chips. With CPU-GPU fusion chips, we do not need to off-load data using a low bandwidth and high latency PCI-Express bus anymore.

The increasing diversity in modern parallel computer architectures adds additional challenges to programmers. They first need to select a system for their problem, and the selection heavily affects performance and programmability. In Chapter 2, we present that design trade-offs in modern compter architectures affect performance and programmability in different ways even for seemingly similar dense numerical computations. Chapters 3, 5, and 6 also highlight the importance of selecting a right programming model and an architecture for a problem of interest. However, identifying a right system for an application of interest requires the deep understanding of the application's computational requirements and the capability of different programming models, system software tools, and architectures. Fortunately, this is not a problem for every parallel programmer. A small number of researchers and project leaders can identify right systems for different classes of applications, and many programmers can follow such guidelines.

Different architectures also require different performance optimization techniques or even different high level algorithms—the GPU implementation in Chapter 2 is an example. Heterogenous systems are even more challenging. Programmers need to assign computational kernels to an appropriate part of the system and also tune different kernels for different architectures. Wide vector units in accelerator architectures improve peak performance per watt, but it is very difficult to achieve a significant fraction of the peak performance in many cases. Vector units in NVIDIA GPUs and Intel architectures are also widely different. In NVIDIA GPUs, multiple threads in a thread group share a wide vector unit (one vector lane per thread), and only the threads in a same execution path can use vector lanes. A vector unit can be partially exploited. In Intel architectures, one thread—without simultaneous multithreading—monopolize a vector unit. In a give cycle, a vector unit (more precisely, a single pipeline stage of the vector unit) is either 100% utilized or 0% utilized. With simultaneous multithreading, a vector unit can be 100% utilized in each cycle if at least one of the threads sharing the vector unit has a vector instruction to execute. Programmers need to consider these differences in software development.

7.2 System Software Support for Nested Irregular Parallelism

There are several unavoidable factors that make parallel programming difficult. For large scale supercomputers and applications with huge matrices and vectors, the high level understanding of data access and communication patterns is crucially important to develop scalable software. Programmers—with the MPI programming model—need to manually control the mapping of computations to computing units and data transfers between computing units. There has been very little success in automating this process. Yet, there are another class of applications—with nested irregular parallelism—that can benefit a lot from system software support. Again, it is extremely difficult for compilers to automatically find the structure of parallelism in such applications. However, if programmers expose the parallelism in their application by marking parallelization points in their code, compilers can easily understand the hierarchy of parallelism without complex dependency analysis. The hierarchical nature of the parallelism in those applications also maps well with the hierarchical memory subsystem of modern parallel computers. The MapReduce programming model demonstrated that proper system software support can significantly reduce programming effort for a class of data-intensive applications. Our experience with Intel TBB reveals another possibility for applications with nested irregular parallelism.

Yet, there are only a very limited number of programs that exploit irregular nested parallelism. For example, Robinson *et al.* [106] presented experimental results using benchmarks with only flat parallelism in their paper introducing Intel TBB. Yet, Intel TBB's main strength is in its support for nested parallelism. Also, IBM implemented LU factorization, Fourier transform, and streaming and random access benchmarks using X10 [5]—which is a HPCS language. HPCS languages' support for nested irregular parallelism and atomic blocks distinguishes those languages from other alternatives (*e.g.* UPC). None of the four benchmarks is suitable to test such features. If there were abundant benchmarks with nested irregular parallelism, they would report the experimental results for those programs as well. We believe that the current scarcity of such programs is not due to the lack of applications with such computational characteristics but more due to other reasons. First, exploiting nested irregular parallelism is too difficult without proper system software support. Second, nested irregular parallelism is more common for commercial or emerging applications than traditional scientific applications, but users of those applications are mostly using desktop PCs with two or four cores. The payoff for using nested irregular parallelism is not very high for such systems. Third, there are not enough success stories to attract more programmers. The core count in desktop PCs will increase fast. Major software and hardware vendors are also working on system software tools—Intel's TBB and Microsoft's TPL for shared memory systems and HPCS languages for large clusters—to support irregular nested parallelism. Missing parts are benchmark programs to test such tools and success stories to entice more programmers. We provide the phylogenetic tree reconstruction code using Intel TBB in this dissertation to fulfill the requirements. In addition, we discuss several ideas to further improve those tools in the remainder of this section.

The work stealing algorithm of Cilk [21] selects a victim thread in random and steals work from the thread's highest level non-empty work queue; the highest level work queue corresponds to the coarsest parallelism. Randomly selecting a victim thread ignores the hierarchical nature of memory subsystem. Stealing work from the highest level work queue reduces communication but often increases memory footprint. Stealing from close threads first can improve performance especially for large scale parallel computers. If a thread steals work from another thread in the same chip, the increase in communication may not be significant. In this case, stealing work from a low level work queue may produce better results in many cases. If a thread needs to steal work from distant thread (especially a thread in another shared memory node), stealing from a high level work queue will be more desirable to reduce communication. A scheduler can consider interconnection network bandwidth utilization and cache usage in making decisions. Our phylogenetic tree reconstruction code serves as a benchmark to test such ideas.

The final goal of our phylogenetic tree construction code is to reconstruct the phylogenetic tree using a leadership class supercomputer. Our code currently considers only inversion, but we are planning to adopt more biologically feasible—but also computationally more expensive—models. Our code is written in a highly extensible form, and we also expect other researchers to use our code as a template to test their algorithms. This involves several issues. We will provide enough parallelism to exploit all the cores in a leadership class supercomputer. However, we want to run our code on desktop PCs or small clusters as well. A supercomputer version of our code will provide over one million-way parallelism, but this will be too much parallelism for a desktop PC. Manually adjusting a degree of parallelism for every target system is not a viable option. Also, if we replace the current pairwise distance computation routine and the median solver with more computationally expensive ones, the structure of parallelism and the amount of computing and communication in each parallelization point change dramatically. The situation is much more problematic if other researchers add their routines to our code. They may not understand the structure of parallelism in our code. Also, different biologists may use data with different characteristics. For example, biologists studying mammalian nuclear genomes may use a small number of large genomes, while biologists studying organellar genomes may use a large number of small genomes. This affects the structure of parallelism. Furthermore, the best parallelization strategy changes based on the communication and thread management cost of a target system. This calls for an auto-tuning approach. Assume that programmers provide abundant parallelism and users provide typical input data sets. If system software can select appropriate parallelization points through an auto-tuning process, this can significantly improve software productivity.

Our phylogenetic tree construction code first reads the input genome data and also updates the pairwise distances between the input genomes. Those data do not change after they are first updated and are read repeatedly. If those data do not fit within cache memory, they need to be read again and again from main memory. In a NUMA system, reading data from local memory is faster than reading data from remote memory. This is even more true for large scale clusters. If system software provides a simple interface for programmers to mark read only and heavily read data and automatically replicates the data and reads the data from the local copy, this can improve performance with the minimum increase in programming complexity.

7.3 Transactional Memory

Data synchronization and nondeterministic execution are major factors that complicate parallel programming—especially for irregular applications. Transactional Memory (TM) cannot completely solve the problems, but TM can significantly alleviate the problems if TM has low performance overhead for most practical cases. Programmers do not need to consider different data synchronization primitives. TM also lowers the pressure to minimize the size of critical sections. Programmers need to consider only transaction level interleavings, and this prunes out a very large fraction of the interleaving scenarios and lowers the degree of nondeterminism. All these combined, TM largely reduces the gap between high-level algorithm design and its actual implementation.

However, software TM incurs high performance overhead if a large fraction of the executed instructions are transactional loads and stores. Our experimental results for the new TM algorithm to compute a minimum spanning forest (Chapter 3) highlight the point. Hardware TM has only moderate performance overhead even for computations with a large number of transactional loads and stores as Dice *et al.* [42] demonstrated with the Sun Rock processor. Yet, hardware TM has several shortcomings. Hardware TM cannot handle transactions that overflow its capacity and does not scale beyond the scalability limit of the cache coherency mechanism. Providing forward progress guarantee in high contention scenarios is more costly for hardware TM as well. Software TM can complement hardware TM, and we consider this as the software TM's role. To achieve high performance, programmers need to reduce the size of very large transactions or decompose a very large transaction to multiple smaller transactions. TM still needs to guarantee correctness regardless of transaction sizes to allow programmers to incrementally tune their code. TM may not provide low performance overhead on large scale clusters for applications with a very low degree of locality. TM still needs to deliver reasonable performance—for applications with hierarchical locality—if transactional loads and stores across shared memory nodes are infrequent. TM also needs to provide forward progress guarantee for extremely high contention scenarios though TM may not provide high performance or scalability for those cases. Software TM can serve this role. Software TM needs to provide mechanisms to guarantee correctness and forward progress while minimizing the interference on the performance of hardware Transactional Memory primitives, and we consider this as an important research topic.

7.4 Hybrid Data Transfer Mechanism

The Cell Broadband Engine architecture's DMA based block data transfer mechanism and the MPI programming model's block data transfer mechanism are effective for communication-intensive applications with high spatial locality and predictable data access patterns. Moving one large data chunk is more efficient than transferring many small data chunks in most parallel computers. This often leads to higher bandwidth utilization as demonstrated in Chapters 2 and 4. We can fine-control data transfers using a block data transfer mechanism to better exploit the locality in data access patterns. Block data transfer mechanisms also have an advantage over cache coherence protocol based data transfer mechanisms in scalability. Cache coherence protocols guarantee the coherency of a cache line by allowing only one thread to own the line for modification, but this incurs additional coherency traffic. This is unnecessary overhead for thread local variables—if we ignore context switching or thread migration. Read only variables do not require a mechanism to guarantee coherency. Computations often consists of multiple phases with a barrier between two phases. In many cases, flushing data at the end of each phase is sufficient to avoid data races. In these cases, block data transfer mechanisms have a performance advantage.

However, using a block data transfer mechanism requires additional coding. The reward for additional coding effort in terms of performance is not significant for small granularity irregular data accesses or infrequently executed parts of the code. A memory subsystem that combines the two different data transfer mechanisms can exploit the strengths of the two in a synergistic way. Programmers can fast prototype their algorithms using only a cache coherence protocol based data transfer mechanism and incrementally tune data transfers using a block data transfer mechanism. IBM released a software based cache coherence library for the Cell Broadband Engine architecture. Using this mechanism for prototyping and using DMA data transfers only when it is needed to achieve high performance can improve productivity for many applications. The NVIDIA Fermi architecture also has a hybrid memory system with both cache memory and local (incoherent) scratch pad memory though its cache memory is only semi-coherent—a cache line needs to be flushed to L2 cache to guarantee coherency. We consider these as precursors of hybrid memory systems to achieve high performance with the minimum increase in programming complexity.

7.5 Future Research Directions

We are interested in extending our parallelization work for the *COGNAC* software package to extreme scale systems. Currently, *COGNAC* is based on inversion distance which compromises flexibility and biological plausibility to limit computational complexity. We are working on designing a more biologically plausible distance metric and algorithms to compute pairwise distances and median genomes for the metric. This will significantly increase the amount of computing to simulate genome data. We plan to address this increase by using a supercomputer with one million-way parallelism (such as the IBM Blue Waters). The programming challenges discussed in this dissertation—such as managing nested irregular parallelism and addressing data synchronization and communication issues—will become significantly more prominent in this scale. Our future research will identify key computational challenges and the role of system software and architectural support in scaling applications with nested irregular parallelism to extreme scale computers.

7.6 Final Remarks

This dissertation presents parallelization and architecture specific performance tuning for various kernels and emerging applications for a spectrum of programming models and architectures. The diversity in programming models and architectures challenges programmers as they need to identify a right set of programming models and architectures for their applications and design and implement algorithms for different systems. However, this dissertation also demonstrates that this diversity can significantly improve software productivity if programmers use right tools in right places. For example, using Transactional Memory largely reduces the difficulty in addressing data synchronization issues in irregular graph algorithms. The hybrid combination of a highly multithreaded system and a MapReduce cluster exploits the strengths of the two in a synergistic way and solves large-scale complex network analysis problems at hand. System software support for nested irregular parallelism—such as Intel TBB—largely reduces the challenges in parallelizing our *COGNAC* software package.

This dissertation also identifies that system software and architectural support have strong potential to significantly reduce the complexity of parallel programming. Understanding future applications and their computational challenges is crucially important to realize the potential, and we provide feedback to the system software and computer architecture communities by providing benchmarks and suggestions especially for system software support for nested irregular parallelism, Transactional Memory, and communication and data transfer mechanisms.

There are several unavoidable factors that complicate parallel programming. Those will remain as tasks for researchers and programmers. Still, we believe that collaborative efforts among the researchers and developers in the application, algorithm, system software, and computer architecture domains can significantly improve the current practice and largely reduce the difficulty. This dissertation—by discussing the relevant issues from an application- and algorithm-centric viewpoint—participates in the collaborative efforts to solve the current parallel computing challenges.

REFERENCES

[1] "AMD CodeAnalyst," 2009. `http://developer.amd.com/cpu/CodeAnalyst`.

[2] "PAPI," 2009. `http://icl.cs.utk.edu/papi`.

[3] "The R project for statistical computing," 2009. `http://www.r-project.org/`.

[4] "Welcome to Apache Hadoop!," 2009. `http://hadoop.apache.org/core`.

[5] "X10-hpcc09," 2009. `http://x10.codehaus.org/hpcc09`.

[6] Adams, M. D. and Kossentini, F., "Jasper: a software-based JPEG-2000 codec implementation," in *Proc. IEEE Int'l Conf. on Image Processing (ICIP)*, (Vancouver, Canada), Sep. 2000.

[7] Albert, R., Jeong, H., and Barabasi, A.-L., "The diameter of the world wide web," *Nature*, vol. 401, pp. 130–131, 1999.

[8] Albert, R., Jeong, H., and Barabasi, A.-L., "Error and attack tolerance of complex networks," *Nature*, vol. 406, pp. 378–382, 2000.

[9] AMD Corporation, *Software Optimization Guide for AMD Family 10h Processors*, 3.06 ed., Apr. 2008.

[10] Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S., "Unbounded transactional memory," in *Proc. Int'l Conf. on High-Performance Computer Architecture (HPCA)*, (San Francisco, CA), Feb. 2005.

[11] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A., "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2006.

[12] Bader, D. A., Agarwal, V., and Kang, S., "Computing transforms on the IBM Cell Broadband Engine," *Parallel Computing*, vol. 35, no. 3, pp. 119–137, 2009.

[13] Bader, D. A. and Cong, G., "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," *J. of Parallel and Distributed Computing*, vol. 66, no. 11, 2006.

[14] Bader, D. A., Cong, G., and Feo, J., "On the architectural requirements for efficient execution of graph algorithms," in *Proc. Int'l Conf. on Parallel Processing (ICPP)*, (Oslo, Norway), Jun. 2005.

[15] Bader, D. A., Kanade, V., and Madduri, K., "SWARM: A parallel programming framework for multi-core processors," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP)*, (Long Beach, CA), Mar. 2007.

[16] Bader, D. A. and Madduri, K., "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *Proc. Int'l Conf. on Parallel Processing (ICPP)*, (Columbus, OH), Aug. 2006.

[17] Bader, D. A. and Madduri, K., "A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms," *Parallel Computing*, vol. 34, no. 11, pp. 627–639, 2008.

[18] Bader, D. A. and Madduri, K., "SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, (Miami, FL), Apr. 2008.

[19] Bienia, C., Kumar, S., Singh, J. P., and Li, K., "The parsec benchmark suite: characterization and architectural implications," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, (Toronto, Canada), Oct. 2008.

[20] Bisciglia, C., Kimball, A., and Michels-Slettvet, S., "Lecture 5: Graph Algorithms & PageRank," 2007. `http://code.google.com/edu/submissions/mapreduce-minilecture/lec5-pagerank.ppt`.

[21] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y., "Cilk: an efficient multithreaded runtime system," *ACM SIGPLAN Notices*, vol. 30, no. 5, pp. 207–216, 1995.

[22] Blundell, C., Devietti, J., Lewis, E. C., and Martin, M. M. K., "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *Proc. Int'l Symp. on Computer Architecture (ISCA)*, (San Diego, CA), Jun. 2007.

[23] Blundell, C., Lewis, E. C., and Martin, M. M. K., "Deconstructing transactional semantics: The subtleties of atomicity," in *Proc. Ann. Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, (Madison, WI), Jun. 2005.

[24] Bocchino Jr., R. L., Adve, V. S., Adve, S. V., and Snir, M., "Parallel programming must be deterministic by default," in *Proc. USENIX conf. on Hot topics in parallelism. (HotPar)*, (Berkeley, CA), Mar. 2009.

[25] Bockenbach, O., Knaup, M., and Kachelriess, M., "Implementation of a cone-beam backprojection algorithm on the Cell Broadband Engine processor," in *Proc. SPIE Medical Imaging*, (San Diego, CA), Feb. 2007.

[26] Bourque, G. and Pevzner, P. A., "Genome-scale evolution: Reconstructing gene orders in the ancestral species," *Genome Research*, vol. 12, pp. 26–36, 2002.

[27] Burt, P. J. and Anderson, E. H., "The Laplacian pyramid as a compact image code," *IEEE Trans. Communications*, vol. 31, no. 4, pp. 532–540, 1983.

[28] Cavalli-Sforza, L. L. and Edwards, A. W. F., "Phylogenetic analysis: models and estimation procedures," *American J. of Human Genetics*, vol. 19, no. 3, pp. 233–257, 1967.

[29] Chakrabarti, D., Zhan, Y., and Faloutsos, C., "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int'l Conf. on Data Mining (SDM)*, (Lake Buena Vista, FL), Apr. 2004.

[30] Charles, J., Jassi, P., Ananth, N. S., Sadat, A., and Fedorova, A., "Evaluation of the Intel Core i7 turbo boost feature," in *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC)*, (Austin, TX), Oct. 2009.

[31] Chaver, D., Prieto, M., Pinuel, L., and Tirado, F., "Parallel wavelet transform for large scale image processing," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, (Ft. Lauderdale, FL), Apr. 2002.

[32] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K., "A performance study of general-purpose applications on graphics processors using CUDA," *J. of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[33] Che, S., Li, J., Sheaffer, J. W., Skadron, K., and Lach, J., "Accelerating compute-intensive applications with GPUs and FPGAs," in *Proc. IEEE Symp. on Application Specific Processors (SASP)*, (Anaheim, CA), Jun. 2008.

[34] Chen, T., Raghavan, R., Dale, J., and Iwata, E., "Cell Broadband Engine Architecture and its first implementation-a performance view," *IBM J. of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[35] Choi, J.-D. and Srinivasan, H., "Deterministic replay of Java multithreaded applications," in *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools (SPDT)*, (Welches, OR), 1998.

[36] Chung, J., Minh, C. C., Carlstrom, B. D., and Kozyrakis, C., "Parallelizing SPECjbb2000 with transactional memory," in *Proc. Workshop on Transactional Memory Workloads (WTW)*, (Ottawa, Canada), Jun. 2006.

[37] Cohen, J., "Graph twiddling in a MapReduce world," *Computing in Science and Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[38] Costa, L. F., Rodrigues, F. A., Travieso, G., and Villas Boas, P. R., "Characterization of complex networks: A survey of measurements," *Advances In Physics*, vol. 56, no. 1, pp. 167–242, 2007.

[39] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D., "Hybrid transactional memory," in *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, (San Jose, CA), Oct. 2006.

[40] Dean, J. and Ghemawat, S., "MapReduce: simplified data processing on large clusters," in *Proc. USENIX Symp. on Operating System Design and Implementation (OSDI)*, (San Francisco, CA), Dec. 2004.

[41] Dice, D., Herlihy, M., Lea, D., Lev, Y., Luchangco, V., Mesard, W., Moir, M., Moore, K., and Nussbaum, D., "Applications of the adaptive transactional memory test platform," in *Proc. ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, (Salt Lake City, UT), Feb. 2008.

[42] Dice, D., Lev, Y., Moir, M., and Nussbaum, D., "Early experience with a commercial hardware transactional memory implementation," in *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, (Washington, DC), Mar. 2009.

[43] Dice, D., Shalev, O., and Shavit, N., "Transactional locking II," in *Proc. Int'l Symp. on Distributed Computing (DISC)*, (Stockholm, Sweeden), Sep 2006.

[44] Faloutsos, M., Faloutsos, P., and Faloutsos, C., "On power-law relationships of the internet topology," in *Proc. ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, (Cambridge, MA), Aug. 1999.

[45] Felsenstein, J., "Evolutionary trees from DNA sequences: a maximum likelihood approach," *J. of Molecular Evolution*, vol. 17, pp. 368–376, 1981.

[46] Fertin, G., Labarre, A., Rusu, I., Tannier, E., and Vialette, S., *Combinatorics of Genome Rearrangements*. MIT press, 2009.

[47] Fitch, W. M., "Toward defining the course of evolution: minimum change for a specific tree topology," *Systematic Zoology*, vol. 20, pp. 406–416, 1971.

[48] Gepner, P., Fraser, D. L., and Kowalik, M. F., "Second generation quad-core Intel Xeon processors bring 45 nm technology and a new level of performance to HPC applications," *Lecture Notes in Computer Science*, vol. 5101, pp. 417–426, 2008.

[49] Guerraoui, R., Kapalka, M., and Vitek, J., "STMBench7: a benchmark for software transactional memory," in *Proc. European Conf. on Computer Systems (EuroSys)*, (Lisbon, Portugal), Mar. 2007.

[50] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K., "Transactional memory coherence and consistency," in *Proc. Int'l Symp. on Computer Architecture (ISCA)*, (Munchen, Germany), Jun. 2004.

[51] Harris, T., Marlow, S., Jones, S. P., and Herlihy, M., "Composable memory transactions," in *Proc. ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, (Chicago, IL), Jun. 2005.

[52] Hartigan, J. A., "Minimum mutation fits to a given tree," *Biometrics*, vol. 29, no. 1, pp. 53–65, 1973.

[53] Helman, D. R. and JáJá, J., "Prefix computations on symmetric multiprocessors," *J. of Parallel and Distributed Computing*, vol. 61, no. 2, pp. 265–278, 2001.

[54] Herlihy, M., Luchangco, V., Moir, M., and Scherer III, W. N., "Software transactional memory for dynamic-sized data structures," in *Proc. Ann. Symp. on Principle of Distributed Computing (PODC)*, (Boston, MA), Jul. 2003.

[55] Herlihy, M. and Moss, J., "Transactional memory: Architectural support for lock-free data structures," in *Proc. Int'l Symp. on Computer Architecture (ISCA)*, (San Diego, CA), May 1993.

[56] Huson, D. H., Nettles, S. M., and Warnow, T. J., "Disk-covering, a fast-converging method for phylogenetic tree reconstruction," *J. of Computational Biology*, vol. 6, no. 3/4, pp. 369–386, 1999.

[57] Huson, D. H., Vawter, L., and Warnow, T., "Solving large scale phylogenetic problems using DCM2," in *Proc. Int'l Conf. on Intelligent Systems for Molecular Biology (ISMB)*, (Heidelberg, Germany), Aug. 1999.

[58] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Nov. 2007.

[59] Ipek, E., Mutlu, O., Martinez, J. F., and Caruana, R., "Self-optimizing memory controllers: A reinforcement learning approach," *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 39–50, Jun. 2008.

[60] ISO and IEC, *ISO/IEC 15444-1: Information technology-JPEG2000 image coding system-part 1: Core coding system*, 2000.

[61] JáJá, J., *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992.

[62] Jimenez-Gonzalez, D., Martorell, X., and Ramirez, A., "Performance analysis of Cell Broadband Engine for high memory bandwidth applications," in *Proc. Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, (San Jose, CA), Apr. 2007.

[63] Kang, S. and Bader, D. A., "Optimizing JPEG2000 still image encoding on the Cell Broadband Engine," in *Proc. Int'l Conf. on Parallel Processing (ICPP)*, (Portland, OR), Sep. 2008.

[64] Kang, S. and Bader, D. A., "An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs," in *Proc. ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, (Raleigh, NC), Feb. 2009.

[65] Kang, S. and Bader, D. A., "Large scale complex network analysis using a hybrid combination of cloud and a highly multithreaded system," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP)*, (Atlanta, GA), Apr. 2010.

[66] Kang, S., Bader, D. A., and Vuduc, R., "Understanding the design tradeoffs among current multicore systems for numerical computing," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, (Rome, Italy), May 2009.

[67] Kang, S., Tang, J., Schaeffer, S. W., and Bader, D. A., "Rec-DCM-Eigen: Reconstructing a less parsimonious but more accurate tree in shorter time." submitted.

[68] Kendall, M. G., "A new measure of rank correlation," *Biometrika Trust*, vol. 30, no. 1, pp. 81–93, 1938.

[69] Kistler, M., Perrone, M., and Petrini, F., "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.

[70] Krishnaswamy, D. and Orchard, M., "Parallel algorithms for the two-dimensional discrete wavelet transform," in *Proc. Int'l Conf. on Parallel Processing (ICPP)*, (Raleigh, NC), Aug. 1994.

[71] Kulkarni, M., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., and Chew, L. P., "Optimistic parallelism benefits from data partitioning," in *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, (Seattle, WA), Mar. 2008.

[72] Kutil, R., "A single-loop approach to simd parallelization of 2-D wavelet lifting," in *Proc. Int'l Conf. on Parallel, Distributed, and Network-Based Processing (PDP)*, (Montbeliard, France), Feb. 2006.

[73] Lee, E. A., "The problem with threads," Tech. Rep. UCB/EECS-2006-1, University of California Berkeley, Jan. 2006.

[74] Lian, C.-J., Chen, K.-F., Chen, H.-H., and Chen, L.-G., "Analysis and architecture design of block-coding engine for EBCOT in JPEG 2000," *IEEE Trans. Circuits and Systems*, vol. 13, no. 3, pp. 219–230, 2003.

[75] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J., "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, Mar. 2008.

[76] Lusk, E. and Yelick, K., "Languages for high-productivity computing: the DARPA HPCS language project," *Parallel Processing Letters*, vol. 17, no. 1, pp. 89–102, 2007.

[77] Madduri, K., Ediger, D., Jiang, K., Bader, D. A., and Chavarría-Miranda, D., "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," Tech. Rep. LBNL-1703E, Lawrence Berkeley National Laboratory, Apr. 2009.

[78] Marathe, V. J., Scherer III, W. N., and Scott, M. L., "Adaptive software transactional memory," in *Proc. Int'l Symp. on Distributed Computing (DISC)*, (Cracow, Poland), Mar. 2005.

[79] McKee, S. A., Wulf, W. A., Aylor, J. H., Klenke, R. H., Salinas, M. H., Hong, S. I., and Weikle, D. A. B., "Dynamic access ordering for streamed computations," *IEEE Trans. Computers*, vol. 49, no. 11, pp. 1255–1271, 2000.

[80] Meerwald, P., Norcen, R., and Uhl, A., "Parallel JPEG2000 image coding on multiprocessors," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, (Ft. Lauderdale, FL), Apr. 2002.

[81] Mehlhorn, K. and Näher, S., "The LEDA platform of combinatorial and geometric computing," *Communications of the ACM*, vol. 38, no. 1, pp. 96–102, 1995.

[82] Meredith, J. S., Alam, S. R., and Vetter, J. S., "Analysis of a computational biology simulation technique on emerging processing architectures," in *Proc. IEEE Int'l Workshop on High Performance Computational Biology (HICOMB)*, (Long Beach, CA), 2007.

[83] Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K., "STAMP: Stanford transactional applications for multi-processing," in *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC)*, (Seattle, WA), Sep. 2008.

[84] Minh, C. C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K., "An effective hybrid transactional memory system with strong isolation guarantees," in *Proc. Int'l Symp. on Computer Architecture (ISCA)*, (San Diego, CA), Jun. 2007.

[85] Montesinos, P., Hicks, M., King, S. T., and Torrellas, J., "Capo: a software-hardware interface for practical deterministic multiprocessor replay," in *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, (Washington, DC), Mar. 2009.

[86] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A., "LogTM: Log-based transactional memory," in *Proc. Int'l Conf. on High-Performance Computer Architecture (HPCA)*, (Austin, TX), Feb. 2006.

[87] Moret, B. M. E., Bader, D. A., and Warnow, T., "High-performance algorithm engineering for computational phylogenetics," *J. of Supercomputing*, vol. 22, pp. 99–111, 2002.

[88] Moret, B. M. E. and Shapiro, H. D., *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science: Computational Support for Discrete Mathematics 15*, ch. An empirical assessment of algorithms for constructing a minimal spanning tree, pp. 99–117. American Mathematical Society, 1994.

[89] Moret, B. M. E., Siepel, A. C., Tang, J., and Liu, T., "Inversion medians outperform breakpoint medians in phylogeny reconstruction from gene-order data," *Lecture Note in Computer Science*, vol. 2452, pp. 521–536, 2002.

[90] Moret, B. M. E., Tang, J., Wang, L.-S., and Warnow, T., "Steps toward accurate reconstructions of phylogenies from gene-order data," *J. of Computer and System Sciences*, vol. 65, no. 3, pp. 508–525, 2002.

[91] Moret, B. M. E., Wyman, S., Bader, D. A., Warnow, T., and Yan, M., "A new implementation and detailed study of breakpoint analysis," in *Proc. Pacific Symp. on Biocomputing (PSB)*, (Big Island, HI), Jan. 2001.

[92] Muta, H., Doi, M., Nakano, H., and Mori, Y., "Multilevel parallelization on the Cell/B.E. for a Motion JPEG 2000 encoding server," in *Proc. ACM Int'l Conf. on Multimedia (MM)*, (Augsburg, Germany), Sep. 2007.

[93] Mutlu, O. and Moscibroda, T., "Enhancing the performance and fairness of shared DRAM systems with parallelism-aware batch scheduling," in *Proc. Int'l Symp. on Computer Architecture (ISCA)*, (Beijing, China), Jun. 2008.

[94] Natarajan, C., Christenson, B., and Briggs, F., "A study of performance impact of memory controller features in multi-processor server environment," in *Proc. Workshop on Memory Performance Issues (WMPI)*, (Munich, Germany), Jun. 2004.

[95] Nei, M. and Kumar, S., *Molecular Evolution and Phylogenetics.* Oxford University Press, 2000.

[96] Nickolls, J., Buck, I., Garland, M., and Skadron, K., "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, pp. 40–53, Mar. 2008.

[97] Nielsen, O. and Hegland, M., "Parallel performance of fast wavelet transform," *Int'l J. of High Speed Computing*, vol. 11, no. 1, pp. 55–73, 2000.

[98] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.0 ed., Jun. 2008.

[99] Olszewski, M., Ansel, J., and Amarasinghe, S., "Kendo: Efficient deterministic multithreading in software," in *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, (Washington, DC), Mar. 2009.

[100] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E., and Purcell, T. J., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, pp. 80–113, Mar. 2007.

[101] Perfumo, C., Sonmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T., and Valero, M., "The limits of software transactional memory (STM):dissecting Haskell STM applications on a many-core environment," in *Proc. ACM Int'l Conf. on Computing Frontiers (CF)*, (Ischia, Italy), May 2008.

[102] Polka, L. A., Kalyanam, H., Hu, G., and Krishnamoorthy, S., "Package technology to address the memory bandwidth challenge for tera-scale computing," *Intel Technology Journal*, vol. 11, no. 3, 2007.

[103] Rafique, N., Lim, W. T., and Thottethodi, M., "Effective management of DRAM bandwidth in multicore processors," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, (Brasov, Romania), Sep. 2007.

[104] Rajwar, R., Herlihy, M., and Lai, K., "Virtualizing transactional memory," in *Proc. Int'l Symp. on Computer Architecture (ISCA)*, (Madison, WI), Jun. 2005.

[105] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D., "Memory access scheduling," in *Proc. Int'l Symp. on Computer Architecture (ISCA)*, (Vancouver, Canada), Jun. 2000.

[106] Robison, A., Voss, M., and Kukanov, A., "Optimization via reflection on work stealing in TBB," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, (Miami, FL), 2008.

[107] Roshan, U. W., Warnow, T., Moret, B. M. E., and Williams, T. L., "Rec-I-DCM3: a fast algorithmic technique for reconstructing phylogenetic trees," in *Proc. Computational Systems Bioinformatics Conf. (CSB)*, (Stanford, CA), Aug. 2004.

[108] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. W., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, (Salt Lake City, UT), Feb. 2008.

[109] Ryoo, S., Rodrigues, C. I., Stone, S. S., Stratton, J. A., Ueng, S., Baghsorkhi, S. S., and Hwu, W. W., "Program optimization carving for GPU computing," *J. of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, 2008.

[110] Saidani, T., Piskorski, S., Lacassagne, L., and Bouaziz, S., "Parallelization schemes for memory optimization on the Cell processor: a case study of image processing algorithm," in *Proc. Workshop on memory performance (MEDEA)*, (Brasov, Romania), Sep. 2007.

[111] Saitou, N. and Nei, M., "The neighbor-joining method: a new method for reconstructing phylogenetic trees," *Molecular Biology and Evolution*, vol. 4, no. 4, pp. 406–425, 1987.

[112] Sankoff, D. and Blanchette, M., "Multiple genome rearrangement and breakpoint phylogeny," *J. of Computational Biology*, vol. 5, no. 3, pp. 555–570, 1998.

[113] Sankoff, D. and El-Mabrouk, N., "Genome rearrangement," in *Current Topics in Computational Molecular Biology*, pp. 135–156, MIT Press, 2002.

[114] Schenk, O., Christen, M., and Burkhart, H., "Algorithmic performance studies on graphics processing units," *J. of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1360–1369, 2008.

[115] Scott, M. L., Spear, M. F., Dalessandro, L., and Marathe, V. J., "Delaunay triangulation with transactions and barriers," in *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC)*, (Boston, MA), Sep. 2007.

[116] Shriraman, A., Spear, M. F., H., H., Marathe, V. J., Dwarkadas, S., and Scott, M. L., "An integrated hardware-software approach to flexible transactional memory," in *Proc. Int'l Symp. on Computer Architecture (ISCA)*, (San Diego, CA), Jun. 2007.

[117] Sweldens, W., "The lifting scheme: A construction of second generation wavelets," *SIAM J. on Mathematical Analysis*, vol. 29, pp. 511–546, Mar. 1998.

[118] Tang, J. and Moret, B. M. E., "Phylogenetic reconstruction from gene-rearrangement data with unequal gene content," in *Proc. Int'l Workshop on Algorithms and Data Structures (WADS)*, (Ottawa, Canada), Jul. 2003.

[119] Tang, J. and Moret, B. M. E., "Scaling up accurate phylogenetic reconstruction from gene-order data," *Bioinformatics*, vol. 19, no. 1, pp. 305–312, 2003.

[120] Taubman, D., "High performance scalable image compression with EBCOT," *IEEE Trans. Image Processing*, vol. 9, no. 7, pp. 1158–1170, 2000.

[121] Tremblay, M. and Chaudhry, S., "A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor," in *Proc. Int'l Solid State Circuits Conf. (ISSCC)*, (San Francisco, CA), Feb. 2008.

[122] Vaidyanathan, P. P., "Quadrature mirror filter banks, m-band extensions, and perfect reconstruction techniques," *IEEE ASSP Magazine*, vol. 4, no. 7, pp. 4–20, 1987.

[123] Vijaykumar, T., Gopal, S., Smith, J. E., and Sohi, G., "Speculative versioning cache," in *Proc. Int'l Conf. on High-Performance Computer Architecture (HPCA)*, (Las Vegas, NV), Jan. 1998.

[124] Volkov, V. and Demmel, J. W., "Benchmarking GPUs to tune dense linear algebra," in *Proc. Int'l Conf. on High Performance Computing and Networking (SC)*, (Austin, TX), Nov. 2008.

[125] von Praun, C., Bordawekar, R., and Cascaval, C., "Modeling optimistic concurrency using quantitative dependence analysis," in *Proc. ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, (Salt Lake City, UT), Feb. 2008.

[126] Watson, I., Kirkham, C., and Lujan, M., "A study of a transactional parallel routing algorithm," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, (Brasov, Romania), Sep. 2007.

[127] Watts, D. J. and Strogatz, S. H., "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, 1998.

[128] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J., "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. Int'l Conf. on High Performance Computing and Networking (SC)*, 2007.

[129] Yang, L. and Misra, M., "Coarse-grained parallel algorithms for multidimensional wavelet transforms," *J. of Supercomputing*, vol. 12, no. 1-2, pp. 99–118, 1998.

[130] Zhang, Y., Rauchwerger, L., and Torrellas, J., "Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors," in *Proc. Int'l Conf. on High-Performance Computer Architecture (HPCA)*, (Las Vegas, NV), Jan. 1998.