12-2009

Static and Dynamic Scheduling for Effective Use of Multicore Systems

Fengguang Song

University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Fengguang Song entitled "Static and Dynamic Scheduling for Effective Use of Multicore Systems." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Xiaobing Feng, Shirley V. Moore, Michael D. Vose, Robert C. Ward

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Fengguang Song entitled "Static and Dynamic Scheduling for Effective Use of Multicore Systems." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra
Major Professor

We have read this dissertation
and recommend its acceptance:

Xiaobing Feng

Shirley V. Moore

Michael D. Vose

Robert C. Ward

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and Dean of the
Graduate School

(Original signatures are on file with official student records.)

Static and Dynamic Scheduling for Effective Use of Multicore Systems

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Fengguang Song

December 2009

Dedication

To my wife, my parents for their constant love, belief, and support.

Acknowledgments

First, I would like to thank my advisor, Dr. Jack Dongarra, for his guidance, motivation, and support during my graduate study in the Innovative Computing Laboratory. Thanks to Dr. Dongarra for providing me with extensive opportunities to participate in various conferences, research forums and workshops. Thanks also to Dr. Dongarra for having regular weekly meetings with me to patiently discuss, review, and guide my dissertation.

Second, I would like to thank my co-advisor, Dr. Shirley Moore. This dissertation could not have been written without Dr. Moore's insightful suggestions, mentorship, encouragement, and belief.

In addition, I would like to thank Dr. Michael Vose, Dr. Robert Ward, and Dr. Xiaobing Feng for agreeing to serve on my graduate committee. I greatly appreciate their time and invaluable advice to this dissertation.

I also wish to thank Dr. Felix Wolf, Dr. Jian Huang, Dr. Bradley Vander Zanden, Dr. Jesse Poore, and Dr. Stacy Prowell for being extremely helpful and supportive.

Last but not least, I am deeply indebted to my family and old friends. I thank my beautiful wife, Lan Lin for her continued moral support through both good times and hard times. I owe thanks to my parents, Song Linbin and Ji Suzhen, for their extraordinary sacrifices over these years, and to my brother, Song Xuguang, to my sister, Song Min, for always being there for me. I thank my old friends, Thara Angskun, Nikhil Bhatia, Zizhong Chen, Peng Du, Zhiao Shi, Gwang Son, Asim YarKhan and Yuanlei Zhang for the friendship life long.

Abstract

Multicore systems have increasingly gained importance in high performance computers. Compared to the traditional microarchitectures, multicore architectures have a simpler design, higher performance-to-area ratio, and improved power efficiency. Although the multicore architecture has various advantages, traditional parallel programming techniques do not apply to the new architecture efficiently. This dissertation addresses how to determine optimized thread schedules to improve data reuse on shared-memory multicore systems and how to seek a scalable solution to designing parallel software on both shared-memory and distributed-memory multicore systems.

We propose an analytical cache model to predict the number of cache misses on the time-sharing L2 cache on a multicore processor. The model provides an insight into the impact of cache sharing and cache contention between threads. Inspired by the model, we build the framework of affinity based thread scheduling to determine optimized thread schedules to improve data reuse on all the levels in a complex memory hierarchy. The affinity based thread scheduling framework includes a model to estimate the cost of a thread schedule, which consists of three submodels: an affinity graph submodel, a memory hierarchy submodel, and a cost submodel. Based on the model, we design a hierarchical graph partitioning algorithm to determine near-optimal solutions. We have also extended the algorithm to support threads with data dependences. The algorithms are implemented and incorporated into a feedback directed optimization prototype system. The prototype system builds upon a binary instrumentation tool and can improve program performance greatly on shared-memory multicore architectures.

We also study the dynamic data-availability driven scheduling approach to designing new parallel software on distributed-memory multicore architectures. We have implemented a decentralized dynamic runtime system. The design of the runtime system is focused on the scalability metric. At any time only a small portion of a task graph exists in memory. We propose an algorithm to solve data dependences without process cooperation in a distributed manner. Our experimental results demonstrate the scalability and practicality of the approach for both shared-memory and distributed-memory multicore systems. Finally, we present a scalable nonblocking topology-aware multicast scheme for distributed DAG scheduling applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since IBM released Power4 (dual cores) in 2001 and Sun Microsystems released UltraSPARC T1 (eight cores) in 2005, there are great numbers of multicore chips implemented by various vendors [Asanovic et al., 2006]. Traditional microarchitectures typically relies on increasing the complexity of the logic, wire, and design to find more Instruction Level Parallelism (ILP) such as out-of-order and speculative instruction executions within a sequential program. But this trend cannot continue any more due to the diminishing returns for large increases in complexity and the exponentially rising processor clock rates [Hammond et al., 1997]. Compared to the traditional microarchitectures, multicore chips have a simple design, higher performance-to-area ratio, and better power efficiency. Thus, hardware architects have changed their course to rely on multicore architectures. Multicore (or "manycore") processors with hundreds of processing cores on a single die are also imminent in the near future.

Both shared-memory and distributed-memory platforms could consist of multicore systems. There are two programming models to develop parallel programs: shared-memory programming model and distributed-memory programming model. This dissertation first depicts what a future shared-memory multicore machine will look like and then proposes a static scheduling method to improve program performance. Next, the dissertation studies how to use the dynamic directed acyclic graph (DAG) scheduling approach to developing new parallel software for both shared-memory and distributed-memory multicore systems.

1.1 Motivation

With the emergence of multicore chips [Le et al., 2007, Golla, 2007, Seiler et al., 2008], future distributed shared memory (DSM) systems will have less powerful processor cores but will have tens of thousands of cores. Performance asymmetry in multicore platforms is another trend due to budget issues such as power consumption and area limitation as well as various degrees of parallelism in different applications [Balakrishnan et al., 2005, Kumar et al., 2004, Kumar et al., 2006]. We call such a system "heterogeneous manycore DSM system" (Fig. 1.1). Processor cores belonging to the same level (e.g., same chip or board) frequently share memory resources. For instance, cores on the same chip may share an L2 or L3 cache.

The shared-memory programming model is capable of attaining the benefits of large-scale parallel computing without surrendering much programmability [Lu et al., 1995]. Using the shared-memory model, a program can be written as if it were running on a large

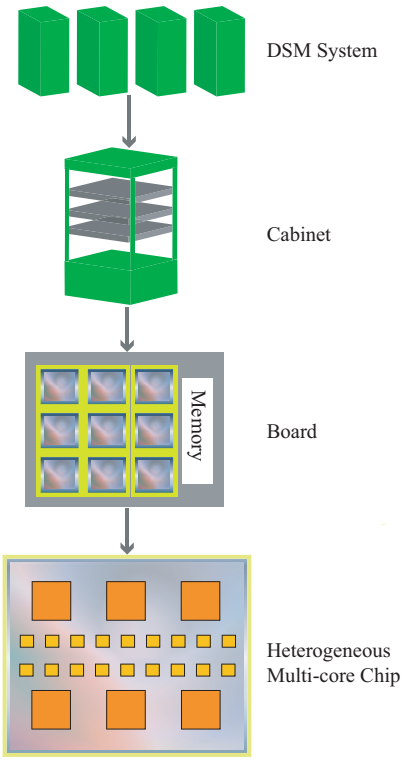


Figure 1.1: Heterogeneous manycore DSM system.

processor count SMP machine. From the perspective of application developers, all processors provide identical performance and the memory access time from each processor is also uniform. This model has been widely accepted and used for a long time. Now if we compare the real architecture and the vision of the architecture from the developers' angle, there is a big gap between them. A number of long-standing assumptions are broken.

- Instead of a uniform memory access time, there are various memory latencies. The immediate result is that placing threads on arbitrary processors may lead to suboptimal performance when there are data accessed in common by threads.
- Heterogeneous cores provide different compute powers. Developers still should be able to write portable programs regardless of different machines.
- When the number of user-level threads is greater than the number of kernel threads, affinity based thread scheduling must be taken into account to maximize the program locality.
- If a number of cores share a certain level of cache, problems may arise due to resource contention.

We hope to find a method to reschedule threads to close the above gap and improve the multithreaded programs' performance. The scheduling method should be automatic and applicable to a variety of general-purpose programs.

Another issue is that multicore chips consist of relatively simple processor cores and will be underutilized if user programs cannot provide sufficient thread level parallelism. It is the developers' responsibility to write high performance parallel software to fully utilize the processor cores. To achieve high performance, we believe that the new parallel multicore software should have the following two characteristics:

- Fine grain threads. We need a high degree of parallelism to keep every processor core busy. Another reason is that a core often has a small-size cache or scratch buffer to work on, which requires developers decompose a task into smaller tasks.

- Asynchronous program execution. When there are many processor cores, the presence of a synchronization point can seriously affect the program performance. And eliminating unnecessary synchronization points can increase the degree of parallelism accordingly.

Therefore, we want to adopt the dynamic DAG scheduling approach to designing new numerical linear algebra libraries for multicore architectures. The dynamic scheduling approach places fine grain computational tasks in a directed acyclic graph and schedules them dynamically depending on data dependence, program locality, and critical path.

1.2 Thesis statement

The main objective of the dissertation is to investigate how to effectively schedule threads to improve program performance on multicore architectures. The dissertation formulates the affinity-based thread scheduling problem on shared-memory multicore systems and proposes a static feedback-directed approach to computing optimized thread schedules to improve the effectiveness on every level of a complex memory hierarchy while keeping load balance. The dissertation also studies the dynamic data-availability driven scheduling approach for fine grain parallel programs and demonstrates the scalability and practicality of the approach on both shared-memory and distributed-memory multicore systems.

1.3 Contributions

The contribution of this dissertation consists of five parts:

- An analytical cache model for shared L2 caches on a multicore processor to predict the number of L2 cache misses when users run multiple threads simultaneously on different cores. The experimental results show that the analytical model has a small average relative error ($< 8.01\%$). In addition, the model provides insight into how cache sharing and cache contention can affect a thread's performance.

- A theoretical foundation for the Affinity Based Thread Scheduling problem on shared-memory multicore systems. After formulating the problem, the dissertation provides an approximation algorithm to compute near-optimal solutions, and an extension of the approximation algorithm to support threads with data dependences.

- A feedback-directed optimization prototype system to automatically instrument binary code and determine optimized thread schedules. The dissertation presents a number of techniques to process large-size affinity graphs. The experimental results show that the feedback directed method is able to reduce the execution time for a variety of applications.

- A distributed deadlock-free algorithm to solve data dependences without process cooperation. A dynamic scheduling runtime system has been implemented and applied to the Cholesky, LU, and QR factorizations. Both theoretical analysis and experimental results have shown that the tiled linear algebra algorithms and the dynamic scheduling method are scalable on both shared-memory and distributed-memory multicore platforms.

- A scalable multicast scheme to support nonblocking data multicasting. The method is proven to be efficient and deadlock-free and is particularly useful for distributed-memory DAG scheduling applications. The performance of the nonblocking multicast is significantly better than the simple flat-tree method and close to vendor-optimized collective MPI operations.

1.4 Original work

To the best of our knowledge, the dissertation includes the following new and original work that no one has done before.

1. First analytical model to predict the number of cache misses for single-application threads on multicore processors with shared caches.

2. First analytical model to estimate the affinity cost of thread schedules.

3. First to tackle the scheduling problem to improve memory effectiveness for all the levels in a memory hierarchy, and first to formulate it as an optimization problem and design approximation algorithms to solve it.

4. First to resolve data dependences without process communication in a distributed manner.

5. First to apply the dynamic data flow scheduling approach to dense linear algebra problems on distributed-memory multicore systems and demonstrate that the approach is scalable by both theoretical analysis and experimental results.

6. A new scalable nonblocking topology-aware multicast scheme for distributed DAG scheduling applications.

1.5 Dissertation outline

The dissertation is organized as follows:

Chapter 2 introduces the related work and compares it to each work described in this dissertation, respectively.

Chapter 3 presents the analytical model to predict the number of cache misses on multicore processors with shared L2 caches.

Chapter 4 formulates the affinity based thread scheduling problem and proposes an analytical cost model as well as an approximation algorithm to solve the problem.

Chapter 5 describes the feedback directed optimization prototype system to automatically instrument user executables and determine optimized thread schedules using Chapter 4's algorithms.

Chapter 6 describes the distributed dynamic scheduling approach to executing numerical linear algebra algorithms.

Chapter 7 presents the scalable nonblocking multicast scheme that is particularly useful for distributed DAG scheduling applications.

Chapter 8 concludes the dissertation and gives directions for future work.

Chapter 2

Related Work

This chapter reviews the related work with respect to modeling time-sharing caches, affinity based thread scheduling, dynamic thread scheduling, and nonblocking multicast.

2.1 Modeling shared caches

Agarwal [Agarwal et al., 1989] develops a cache model that is driven by a small number of parameters such as start-up effect, non-stationary behavior, intrinsic interference, and extrinsic interference. The extrinsic interference considers multiprogramming as an additional source of cache misses and allows for estimating cache performance for round-robined multiprogramming processes. The model introduces a notion of *carry-over set* to denote the cache blocks left behind due to a context switch. But the hypotheses of uniform distribution of both program blocks and interference misses between potential colliding blocks are sometimes not accurate in practice. Thiébaut [Thiébaut and Stone, 1987] introduces an analytical model to estimate the cache-reload transients. A reload transient occurs when an out-switched process resumes after it is rescheduled by the kernel. The model uses four parameters: footprint of process A, footprint of process B, number of cache sets, and cache set associativity. Both models are able to estimate the effect of process swapping (i.e., space-sharing), but they are not suitable for modeling simultaneous and concurrent accesses to a

shared cache (i.e., time-sharing) since process swapping allows only one process to execute at a time.

Suh [Suh et al., 2002] proposes to use a set of hardware counters which are fully-associative counters, way-counters, and set-counters to monitor the *marginal-gain* in cache hits for a set of processes. The scheme assumes the LRU replacement policy and uses `counter(i)` to denote the number of references to the i-th most recently referenced block. The author then introduces an analytical model to combine different sets of counters to estimate the overall miss rate if multiple processes execute simultaneously. The model supports both time-sharing and space-sharing between processes, but it assumes that there is no datum shared between processes. Chandra [Chandra et al., 2005] introduces an inductive probability model using circular sequences to predict cache interference between co-scheduled threads on multicore chips. A circular sequence profile is a collection of *cseq(d, n)* counters each of which denotes the number of series of n cache accesses to d distinct cache lines. The circular sequence profile provides more information than the marginal gain counters and can be used to derive the corresponding marginal gain counters. The probability model also assumes that the co-scheduled threads are from different processes and have disjoint address spaces. However, there are many scientific applications that use the shared-memory programming model (e.g., OpenMP programs). Our model extends Chandra's work and takes into account the factor of memory sharing between threads. Chapter 3 targets this new problem of shared-cache modeling for shared-memory programs.

2.2 Affinity based thread scheduling

A lot of research work has proposed ways of reorganizing data structures and altering programs to improve the memory access efficiency.

Augmented Task Graph considers both computation and communication among tasks for distributed-memory systems [El-Rewini et al., 1994]. The dissertation extends this approach to shared-memory systems by introducing the "affinity graph" to describe the data-reuse relationship between different threads (or tasks). The dissertation defines the metric of data reuse as the number of addresses accessed in common by two threads.

Philbin designs a user-level thread library to improve cache locality using fine-grained threads [Philbin et al., 1996]. All the data-independent units of computation implied in the sequential program are created as fine-grained threads all at once. When a thread is created, a hint of the starting addresses of the accessed arrays must be provided as an argument. After thread creation, the thread library reschedules the threads at runtime to reduce the number of L2 cache misses. This method only works for sequential single-thread programs.

Yan develops a runtime library to maximize data reuse [Yan et al., 2000]. His approach is more generic and can be applied to parallel programs on SMP machines. He adopts an adaptive scheduler to achieve load balance between processors. Similar to Philbin's approach, Yan also uses the starting addresses of arrays as hints to determine an optimal schedule. Pingali uses locality groups to restructure computations for a variety of applications but requires hand-coded optimizations [Pingali et al., 2003]. In contrast, we use a binary instrumentation tool to analyze memory trace and automatically acquire more precise affinity information between threads. Such precise information is critical for thread scheduling on a NUMA distributed shared memory system. And the overhead to execute an optimized schedule is less than that of a runtime system by using a feedback-directed optimization method.

Ding attempts to improves program locality through trace-driven computation regrouping [Ding and Orlovich, 2004]. He develops a trace-driven tool to measure the exact control dependence between instructions and applies techniques of memory renaming and reallocation. Due to the expense of scheduling individual instructions, Ding's method is limited to small fragments of a few kernels. And it is only applicable to sequential programs.

Similar to our approach, Träff applies the graph partitioning technique to solve the MPI process mapping problem [Träff, 2002]. The author designs a framework to compute an

optimal MPI process placement to minimize the message passing cost. Pichel formulates sparse matrix-vector product as a graph problem where each row of the sparse matrix represents a vertex [Pichel et al., 2004, Pichel et al., 2005]. Pichel's method works effectively for both SMP and ccNUMA DSM systems, but is limited to the SpMV application.

Affinity loop scheduling minimizes the cache miss rate by allocating loop iterations to the processor whose cache already contains the necessary data [Markatos and LeBlanc, 1994]. The affinity loop scheduling method emphasizes loop iteration assignment and assumes that the loop iterations have an affinity to particular processors. A typical use of this method is to schedule a parallel loop that is nested within a sequential loop. Unlike affinity loop scheduling, our work reorders and parallelizes the inner loop iterations before assigning them to particular processors. Furthermore, we use more generic fine-grained threads as the scheduling unit rather than loop iterations.

2.3 Dynamic thread scheduling

Most of the work that uses dynamic thread scheduling has focused on shared-memory systems. Cilk is a multithreaded language that generalizes the semantics of C and uses the "work-stealing" scheduling algorithm that is provably efficient in terms of time, space, and communication [Blumofe and Leiserson, 1999, Frigo et al., 1998]. In the Cilk runtime system, each processor has a *deque* to store threads. An idle processor picks up a thread by removing the thread at the bottom of the deque. It works on the thread until the thread spawns, stalls, dies, or enables a stalled thread. If its deque is empty, the processor starts stealing threads from other processors randomly. But the Cilk programming model is limited to solving recursive problems. And programmers must explicitly define barriers with `sync` to specify data dependences for the tasks located in the same recursion level.

Buttari et al. design a collection of tiled linear algebra algorithms for multicore machines [Buttari et al., 2009, Kurzak et al., 2008]. Their algorithms use the block data layout and schedule fine-grained tasks dynamically. Recently they extend their work and develop a new library called PLASMA for a richer set of linear algorithm algorithms. Although using

dynamic scheduling, their initial implementation defines the data dependence relationship for each algorithm in hardwired code.

OpenMP provides a standard interface for shared-memory parallel programming. Programmers can insert OpenMP directives to their sequential programs to add parallelism incrementally. OpenMP is particularly useful for programs with regular parallel loops. The new OpenMP 3.0 addresses the irregular parallelism issue and proposes to use the task-queuing model and dynamic sections to support `while` loops and recursive functions [Ayguade et al., 2009]. But when declaring parallel loops or parallel sections, it is the user's responsibility to guarantee that there is no dependence between them.

Intel Thread Building Blocks (TBB) is a C++ library that supports shared-memory parallel programming [Reinders, 2007]. Users specify "tasks" to construct parallel programs and let the runtime system schedule the tasks onto processor cores. TBB provides templates for common patterns such as loops, pipelines, and nested parallelism. TBB uses the work-stealing scheduler and is best suited for recursive algorithms. Like Cilk, TBB requires users define barriers to specify dependences explicitly. By contrast, our scheduling runtime system takes fully responsibility for detecting data dependences and allows users to ignore data dependences while writing parallel programs.

SMP Superscalar (SMPSs) is a parallel programming environment for multicore architectures [Perez et al., 2008]. Programmers add pragmas to the same sequential C code to identify atomic tasks. Then SMPSs compiles the sequential C program and links it with the runtime system, and executes the program in parallel. Similar to our dynamic scheduling approach, SMPSs is able to analyze data dependences at runtime. Instead of using compiler technology, we replace the basic linear algebra routines by a task-based library to execute programs in parallel automatically. Out work is more focused on designing scalable software for distributed-memory systems. The same code is able to work on both shared-memory and distributed-memory multicore systems in an efficient and scalable manner. Chapter 6 describes our dynamic task scheduling work.

2.4 Nonblocking multicast

MPICH-G2 uses depth information to represent where an MPI process is located in a computational Grid [Karonis et al., 2003]. There are four levels of depths: *wide area*, *local area (or site)*, *system area*, and *node-specific area*. For instance, a cluster in a site has a depth of 3, and a multicore node has a depth of 4. On every level i, all the MPI processes are partitioned into groups for which the same group of processes can communicate through the ith network level. Groups are represented by different colors. Two processes on a particular level are assigned the same color if they can communicate at that level. The topology table consists of a number *depths* of rows and a number *processes* of columns. Table[d, p] defines a color for process p on network level d. The table is global for the whole grid and must be accessible by every process. Our topology ID representation has a distributed compact table and requires much less space.

Plaxton designs a distributed data structure of neighbor table to maintain and access nearby shared objects in a distributed environment. The distributed environment is very dynamic and large numbers of objects are inserted, copied, and deleted constantly [Plaxton et al., 1997]. In Plaxton's access scheme, every different object has a "virtual" balanced tree embedded into the network. With the embedded tree, accessing an object can be realized by checking the node's local memory and the node's subtree, followed by forwarding the access request to its parent. Plaxton's neighbor table is the key element for the different trees for different objects. Section 7.5 describes the definition of the neighbor table. Our work is an extension to Plaxton's neighbor table where we build routing tables for processes instead of nodes. Plaxton assumes each node has a label that is independently and uniformly distributed at random. In contrast, we assign each node a topology ID to enable network topology awareness. Since a user's processes occupy a subset of all the nodes on the system, we modify the table-building method to allow empty entries (or "holes") in routing tables. Furthermore, Plaxton's method is intended to locate a named object by tracking the tree from a leaf to the root. Differently, we design a scheme to do multicasting from the root to a set of leaves by extending Plaxton's table to a new routing table.

Wu [Wu and Sheng, 2005] designs a prefix-based multicasting scheme for irregular networks. Each outgoing channel of a node is assigned a label. A multicast packet is first forwarded up to the longest common prefix of the multicast destinations and then forwarded down to leaves along the spanning tree. The whole system is based on a single spanning tree. In our multicast method, every process has its own spanning tree.

Bayeux uses the Tapestry structure to provide an application-level multicast scheme for streaming multimedia applications [Zhao et al., 2001, Zhuang et al., 2001]. Bayeux builds a distribution tree based on four control messages: *JOIN*, *LEAVE*, *TREE*, *PRUNE*. To construct a distribution tree, the source server must advertise the session information first. Then the clients have to join the session explicitly to form the distribtuion tree. Panda proposes a hierarchical leader based approach to support one-to-many multicasting [Panda et al., 1999]. The set of nodes are grouped into subsets explicitly so that each subset is represented by a leader. Banerjee also proposes a hierarchical clustering method to multicast data stream to large receiver sets [Banerjee et al., 2002]. Unlike the above methods, our method does not construct distribution trees or hierarchies explicitly, but simply uses the implied topology-aware spanning trees to do multicasting.

Chapter 3

Level-2 Cache Modeling for Multicore Processors

3.1 Introduction

Cache performance plays an important role in software performance. With the increasing gap between memory and CPU speeds, it is essential to utilize the cache to its full potential. Chip Multi-Processing (CMP) (i.e., multicore) architectures have been developed to enhance performance and power efficiency through the exploitation of both instruction-level and thread-level parallelism. Some CMP architectures share an on-chip L2 cache among cores, and others have private L1/L2 caches. As described in the prior work by Fedorova [Fedorova et al., 2006], an L2 cache miss penalty can be as high as 200-300 cycles while an L1 cache miss only costs a few cycles. Poor L2 cache behavior can dramatically increase the amount of off-chip communication and degrade the overall performance. Thus, our work is focused on modeling the behavior of the on-chip shared L2 cache.

For multi-threaded programs, a shared L2 cache allows higher utilization of the cache as one thread can reuse the same data loaded previously by another thread. Such reuse reduces power consumption and avoids duplicating hardware resources. However, parallel

The material from this chapter was published in the 36th International Conference on Parallel Processing (ICPP 2007) [Song et al., 2007b].

threads often interfere with each other and contend for accesses to the shared L2 cache, leading to suboptimal performance. This chapter presents an analytical model to predict the number of L2 cache misses for shared-memory scientific applications. By analyzing the L2 cache trace which is recorded when just a single thread is running, the model is able to predict the number of misses if users run the thread together with other threads on the remaining cores. The model assumes there is one thread on each core.

Considering the characteristics of thread-parallel programs from scientific computation, nearly all threads are homogeneous. That is, each thread works on the same task in parallel and has similar temporal behavior. Modeling the effect of shared memory accesses leads to a more powerful model that can predict the number of L2 cache misses for threads not only from distinct processes, but also from a single process.

The scheme presented in this chapter classifies cache misses into three types: *compulsory misses*, *capacity misses on shared data*, and *capacity misses on private data*. The terms *shared* and *private* denote whether the data are referenced by more than one thread (shared) or by a single thread (private). For instance, threads from different processes usually have disjoint memory accesses. We model the above three types of misses with three different methods: an average value for modeling compulsory misses, a probability method for modeling capacity misses on private data, and an effective cache space and a probability method for modeling capacity misses on shared data. The goal is to model the L2 cache behavior by discovering the causes a cache hit developing into a cache miss as well as a cache miss turning into a cache hit.

We validate the model using the cycle-accurate simulator SESC [Renau et al., 2005]. Three scientific programs have been implemented to verify the model: matrix multiplication using three nested loops, blocked matrix multiplication, and sparse matrix-vector product taking as input sparse matrices from Matrix Market [Boisvert et al., 1996]. The average relative error of the model is between 2% and 8%.

3.2 Methodology overview

The model uses the stack processing technique to estimate the number of L2 cache misses given an L2 cache trace. However, the L2 cache trace might be changed if users run a thread together with other threads due to the shared cache access. Therefore, the model must be able to predict how a thread's trace is affected by the other threads (we call it "interference"). We employ the technique of circular sequence profiling to realize the prediction. Note that we still need the stack processing technique to derive the number of cache misses from the predicted trace.

3.2.1 Stack processing technique

Gecsei introduces a technique called "stack processing" to evaluate storage hierarchies that use stack algorithms as a replacement policy [Mattson et al., 1970]. A storage hierarchy consists of multiple levels of devices that are partitioned into *pages*. The input to the model is a *page trace* $x_1, x_2, \ldots, x_n$, where x_i is the page number accessed by the program. It is possible to apply the technique to any level of the storage hierarchy as long as there is a corresponding trace. We call a trace the *block trace* if we are examining caches.

Assume a fully-associative cache has $\mathcal{C}$ lines (or ways). It is easy to see that under LRU the cache contains the $\mathcal{C}$ most recently used lines at any time `t`. Even if we increase the cache size to $\mathcal{C}+1$, $\mathcal{C}+2$, ..., the set of $\mathcal{C}$ lines are still stored in the cache. This property is called the "inclusion property" [Mattson et al., 1970]. Because of the inclusion property, the content of the cache at time t is able to be represented as an LRU stack:

$$\mathcal{S}^{(t)} = \{s^{(t)}(1), s^{(t)}(2), \cdots, s^{(t)}(\mathcal{C})\}, \text{ where}$$

$$s^{(t)}(i) = Blocks^{(t)}(\mathcal{C} = i) - Blocks^{(t)}(\mathcal{C} = i-1).$$

$Blocks(m)$ denotes the set of lines contained in a cache of size m. If a cache line x was referenced before, the position $\triangle$ of line x (counted from the top of the stack) is called "stack distance". Let `counter(`$\triangle$`)` accumulate the number of times the stack distance $\triangle$

appears in the page trace. Such a set of counters forms a so-called *stack distance profile*. For instance, `counter(1)` counts the number of hits in the most recently used line, `counter(`$\mathcal{C}$`)` counts the number of hits in the least recently used line, and `counter(`$\mathcal{C}+1$`)` counts the number of cache misses.

Our model assumes a fully associative cache that uses the LRU algorithm and thus has no conflict misses. It has been shown that set-associative cache miss rates are related to fully associative ones and a model using Bayes rule is able to make accurate predictions [Hill and Smith, 1989]. In addition, when the size of set-associativity is large, set-associative caches often have a miss rate comparable to fully associative caches.

A stack distance profile is sufficient to estimate the number of misses for a particular cache capacity. However, a page trace is prone to change because of other threads running on the remaining cores. Hence we must acquire more information to model the possible interferences from the other threads. The concept of *circular sequence profile* is introduced by Chandra [Chandra et al., 2005] and has successfully modeled the interference effect from other processes in distinct address spaces. Note that we can still deduce a stack distance profile from a circular sequence profile.

3.2.2 Circular sequence profile

A *circular sequence* is a sequence of cache lines $x_1, x_2, \cdots, x_n$, where $x_1 = x_n$ and x_1 does not appear anywhere in the middle of the sequence except for the beginning and the end positions [Chandra et al., 2005]. It is possible that other circular sequences exist in the sequence if one cache line appears several times in the middle. For instance, the trace in Fig. 3.1 contains five circular sequences. We use CSEQ(d, n) to denote a set of circular sequences, in which each sequence is of length n and has d distinct cache lines:

$$\text{CSEQ}(d, n) = \{\alpha \,|\, \alpha \text{ is a circular sequence that accesses n lines and has d distinct lines}\}.$$

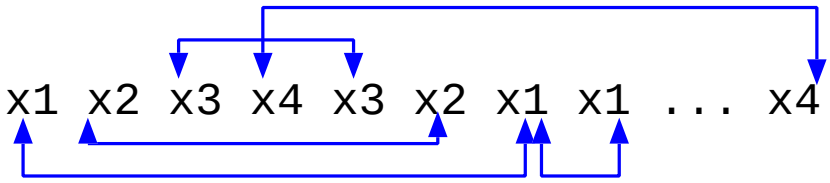


Figure 3.1: An example of a cache block trace containing five circular sequences.

Different d and n define a different circular sequence set. In practice we use a counter to record the number of elements in a nonempty CSEQ(d, n) set. It is denoted as $|\text{CSEQ}(d, n)|$. Each CSEQ(d, n) has a counter and the list of counters forms a circular sequence profile.

We extend the SESC simulator to collect an L2 cache trace that consists of L2 cache lines sent from all processor cores. In the trace file, each cache line is written in the form of `PhysicalAddress:CoreId:VirtualAddress`. The second field `CoreId` helps keep track of a specific thread's trace, and the third field `VirtualAddress` is used by our model to distinguish shared data accesses from private data accesses.

To obtain circular sequence profiles, we use a scan process to analyze the L2 cache trace. Figure 3.2 shows the process's corresponding C++ program. Associative map `addr_map` records a physical address and the position of its most recent appearance in the trace, array `compulsory` counts the total number of compulsory misses for each core, and `cseq_shared`, `cseq_private` are circular sequence set counters for shared and private data, respectively. The analysis process outputs not only compulsory misses for each core, but also circular sequence profiles for shared data $cseq^{shared}(d, n)$ and private data $cseq^{private}(d, n)$. Based on these three components, the model is able to estimate the number of cache misses when running multiple threads simultaneously.

3.3 Modeling strategy

We use the most well-known "three Cs" model (compulsory, capacity, and conflict misses) to classify cache misses. For simplicity, we only consider fully associative caches and thus there is no conflict miss in the model. The model takes as input a single thread's circular sequence profile and estimates the number of misses if the thread had run together with

```
map<paddr, pos> addr_map;
pos = 1;
while(not end of the trace file) {
  read into paddr, coreid, vaddr;
  if(addr_map[paddr] == 0) {
    addr_map[paddr] = pos;
    compulsory[coreid]++;
  } else {
   n = pos - addr_map[paddr] + 1;
   d = get_num_distinct(addr_map[paddr], pos-1);
   addr_map[paddr] = pos;
   if(is_shared(vaddr))
     cseq_shared[coreid][n][d]++;
   else
     cseq_private[coreid][n][d]++;
  }
  pos++;
}
```

Figure 3.2: The C++ program to scan the L2 cache trace to obtain circular sequence profiles and the number of compulsory misses for each processor core.

other threads. Note that the model always measures and predicts the performance of this single thread. Since the model does not consider simultaneous multi-threading (SMT) on processor cores and always runs one thread on each core, this chapter interchangeably uses "thread" and "core".

Suppose we know of a thread's circular sequence profile, then we can derive the number of cache misses by the following expression:

$$misses = compulsory + \sum_{d>\mathcal{C}} \sum_{n>d} |CSEQ(d,n)|.$$

When users run a thread concurrently with other threads on different cores, the trace of that thread will be affected by references from the other threads. We divide L2 cache references into two types based on their addresses: references to shared data, and references to private data. Instead of using a single circular sequence profile $cseq(d,n)$ for each thread, we introduce $cseq^{private}(d,n)$ and $cseq^{shared}(d,n)$. For instance, consider two sequences:

$ABCDA$ where A is shared and $ZBCDZ$ where Z is private. The first sequence increases the counter $cseq^{shared}(4,5)$, while the second increases $cseq^{private}(4,5)$.

For a given thread, different types of references are affected differently by the other threads. For instance, when a shared datum is accessed by two threads, the cache line previously loaded by one thread can save the other from reloading it. Therefore, (i) an original compulsory cache miss might become a hit. The second type is that (ii) the number of capacity misses on private data should increase because a previous hit may become a miss due to interferences from other threads. Finally, predicting the number of capacity misses on shared data is much more complicated: (iii) a cache miss on shared data may become a hit because the other thread already loaded the data, meanwhile (iv) a cache hit on shared data may become a miss owing to the other threads' interference.

The following subsections describe the methods to predict the above four types of misses respectively. $Misses_{new}^{(co)}$ denotes the predicted number of compulsory misses, $Misses_{new}^{(pr)}$ denotes the predicted number of capacity misses on private data, and $Misses_{new}^{(sh)}$ denotes the number of capacity misses on shared data (including types of iii and iv). Furthermore, the total number of L2 cache misses is the sum of $Misses_{new}^{(co)}$, $Misses_{new}^{(pr)}$, and $Misses_{new}^{(sh)}$.

3.3.1 Modeling the compulsory misses

To determine how many compulsory cache misses become cache hits accurately is dependent upon the relative speed of the concurrent threads and how much their working sets overlap. Given thread $t0$, thread $t1$, and shared data accesses b_1, b_2, b_3, b_4, thread $t0$ will have four compulsory misses if it is running alone. With thread $t1$ running, thread $t0$ misses might become fewer if thread $t1$ loads some of the data, or remain to be four if thread $t1$ always lags behind thread $t0$. It is hard to provide a precise prediction unless we know more detailed information.

Since the model is focused on homogeneous threads, it is reasonable to assume that the shared data are loaded evenly by the participating threads. This assumption has been validated by the experiments and most of the time the relative error for the compulsory miss estimate is less than 15%. Figure 3.3 shows an example that launches two threads to

compute $C = A \times B$ using a block data distribution. Matrix B is shared by thread 0 and thread 1. From the perspective of thread 0, around half of its compulsory misses on matrix B may be loaded by thread 1.

We introduce $F_{m2h}^{(co)}$ to denote the fraction of a thread's compulsory cache misses that will become cache hits. Here *co* stands for "compulsory".

$$F_{m2h}^{(co)} = \frac{\text{Overlapped Blocks}}{TotalBlocks \times NumCores}.$$

The fraction of compulsory misses that remain to be compulsory misses $F_{miss}^{(co)}$ is as follows:

$$F_{miss}^{(co)} = 1 - F_{m2h}^{(co)}.$$

Thus the predicted number of compulsory misses for running the thread concurrently with other threads is expressed as:

$$Misses_{new}^{(co)} = Misses_{old}^{(co)} \times F_{miss}^{(co)},$$

where $Misses_{old}^{(co)}$ is the original number of compulsory misses when the thread is running alone.

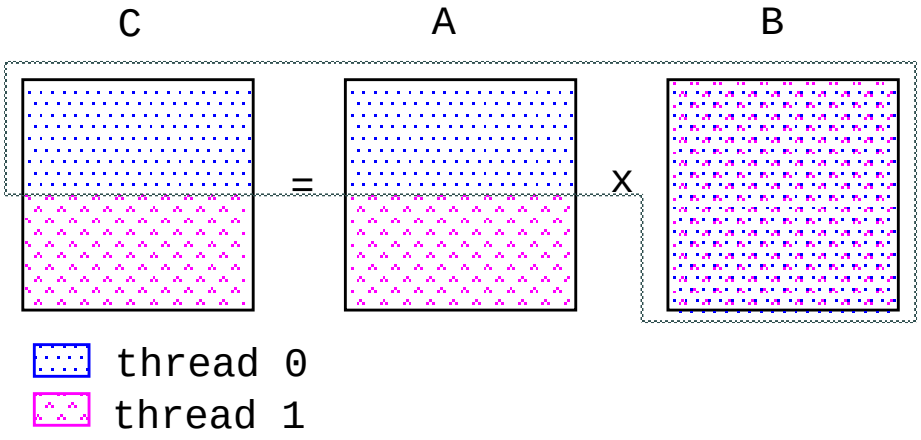


Figure 3.3: Two threads compute matrix multiplication of $C = A \times B$ using the block data distribution. For thread 0, half of its compulsory misses on matrix B could be saved by data loading of thread 1 (i.e., $F_{m2h}^{(co)} = \frac{1}{(.5+.5+1)\times 2} = \frac{1}{4}$).

3.3.2 Modeling the capacity misses on private data

It is easy to see that a capacity cache miss on private data is still a cache miss regardless of whether a thread is running alone or with another thread. But a cache hit may become a cache miss because references from other threads will likely stretch out the circular sequence too much. Figure 3.4 illustrates how a cache hit could become a miss for a cache of size $\mathcal{C} = 4$. The sequence at the bottom is likely to happen if we run thread 0 and thread 1 together. At this time, the second reference to $a1$ is now becoming a cache miss. Therefore, the predicted number of capacity misses on private data is equal to the sum of the original capacity misses and some original hits which turn into misses.

Let thread 0 and thread 1 run in parallel on two different cores. $CSEQ(d, n)$ corresponds to the cache hits of thread 0 if $d \in [1, \mathcal{C}]$. Note that $CSEQ_0(d, n)$ is a set of circular sequences with length n and d distinct addresses. During the time thread 0 is accessing its n addresses in L2, thread 1 is also accessing the shared L2 cache. The references from thread 1 may insert an extra Δd distinct addresses into the circular sequence. When $d + \Delta d > \mathcal{C}$, thread

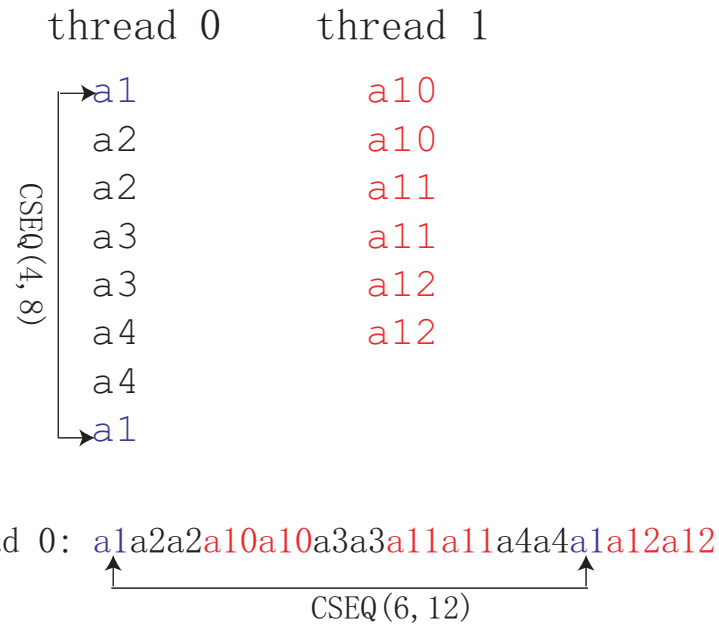


Figure 3.4: A cache hit of thread 0 becomes a miss because of references from thread 1.

0's cache hit develops into a miss. For simplicity, we assume all the references inserted are different from those in the original sequence.

We use $Prob_{h2m}^{(pr)}$ to denote the interference probability for which a cache hit on private data becomes a cache miss. The modeling method for private data is an extension to the technique developed by Chandra et al. [Chandra et al., 2005]. We apply the technique to the private-data circular sequence profile of a thread using $Prob_{h2m}^{(pr)}(cseq^{(pr)}(d,\bar{n})) = \sum_{\hat{d}>\mathcal{C}-d} Prob(seq(\hat{d},\bar{n}))$, where $d \leq \mathcal{C}$ and $\bar{n}$ is the average length of sequences with d distinct addresses. The following step 2 shows how to compute $\bar{n}$. Since we are concerned with homogeneous threads, the model scans the same trace to compute the interference probability $Prob(seq(\hat{d},n))$. The inductive probability function used by [Chandra et al., 2005] is more complex and essentially exponential. In our algorithm, $\sum_{\hat{d}>\mathcal{C}-d} Prob(seq(\hat{d},\bar{n}))$ is computed by scanning the trace file to find the frequency of sequences with length $\bar{n}$ and greater than $\mathcal{C}-d$ distinct addresses. It has a linear time complexity of $O(TraceSize)$.

Our model takes as input the private-data circular sequence profile $CSEQ^{(pr)}(d,n)$ and works as follows:

1. Compute the total number of capacity misses when a single thread is running:

$$Misses_{old}^{(pr)} = \sum_{d>\mathcal{C}} \sum_{n>d} |CSEQ^{(pr)}(d,n)|$$

2. Compute the number of cache hits which become misses:

   for d *= 1 to* $\mathcal{C}$ *do*

$$total\_num = \sum_{n>d} |CSEQ^{(pr)}(d,n)|$$
$$\bar{n} = \frac{\sum_{n>d} \left(|CSEQ^{(pr)}(d,n)| \times n\right)}{total\_num}$$
$$Prob_{h2m}^{(pr)}(d,\bar{n}) = \sum_{\hat{d}>\mathcal{C}-d} Prob(seq(\hat{d},\bar{n}))$$
$$\Delta Misses^{(pr)} += total\_num \times Prob_{h2m}^{(pr)}$$

   end for

3. Finally, compute the predicted number of capacity misses on private data:

$$Misses_{new}^{(pr)} = Misses_{old}^{(pr)} + \Delta Misses^{(pr)}$$

3.3.3 Modeling the capacity misses on shared data

The number of capacity misses happening on the shared data is much more difficult to model than the compulsory misses and the capacity misses on private data. We need to further partition the shared-data circular sequence profile $CSEQ^{(sh)}(d, n)$ into two subcategories: cache hits (circular sequences with $d \leq \mathcal{C}$) and cache misses (circular sequences with $d > \mathcal{C}$). Likewise, cache hits may become cache misses because references from other threads stretch out the sequence length, and cache misses may become cache hits because other threads already loaded the shared data into the L2 cache. To model the two different subcategories, we adopt two different approaches, respectively.

Cache hits on shared data become cache misses

A thread is not able to occupy all the L2 cache lines when it is running concurrently with other threads. A portion of the cache will contain data from the other threads. Since all threads have similar temporal behavior, we assume that the effective cache size $\mathcal{C}_{eff}(t_0)$ of thread t_0 is proportional to the percentage of its footprint size to the overall footprint size:

$$\mathcal{C}_{eff}(t_0) = \frac{|footprint(t_0)|}{|\bigcup_i footprint(t_i)|} \times \mathcal{C}.$$

The number of additional cache misses $Misses_{h2m}^{(sh)}$ is computed by applying $\mathcal{C}_{eff}$ to the circular sequence profile of the concerned thread:

$$Misses_{h2m}^{(sh)} = \sum_{d=\mathcal{C}_{eff}+1}^{\mathcal{C}} \sum_{n>d} |CSEQ(d, n)|.$$

Cache misses on shared data become cache hits

To consider another situation where capacity misses on shared data turn into cache hits, we use the same idea as in predicting the compulsory misses. If m cache lines are accessed by n threads in common, the model assumes each thread will load $\frac{m}{n}$ lines. Therefore the fraction $F_{m2h}^{(sh)}$ of capacity misses that become hits on shared data is expressed as:

$$F_{m2h}^{(sh)} = 1 - \frac{1}{\text{Number of Threads}},$$

and the reduced number of capacity misses is equal to:

$$Misses_{m2h}^{(sh)} = Misses_{old}^{(sh)} \times F_{m2h}^{(sh)}.$$

By the above $Misses_{h2m}^{(sh)}$ and $Misses_{m2h}^{(sh)}$, we can now estimate the number of capacity misses on shared data:

$$\begin{aligned} Misses_{new}^{(sh)} &= Misses_{old}^{(sh)} - Misses_{m2h}^{(sh)} + Misses_{h2m}^{(sh)} \\ &= Misses_{old}^{(sh)} \times \frac{1}{\text{Number of Threads}} + Misses_{h2m}^{(sh)} \end{aligned}$$

Finally, summing up $Misses_{new}^{(co)}$, $Misses_{new}^{(pr)}$, and $Misses_{new}^{(sh)}$ gives the predicted total number of L2 cache misses.

3.4 Experimental results

The implementation of the model consists of a tool analyzing the L2 cache trace to create circular sequence profiles for each core and a library implementing the analytical model. We validate the model using three examples typical of scientific computing. All three experiments perform double-floating point operations on matrices/vectors that are stored contiguously in memory. We use the simple 1-D block data distribution to allocate tasks to two threads. The three experiments are:

- Dense matrix multiplication using three nested loops. We denote it as `dgemm`.
- Dense matrix multiplication using the tiling technique. The tile size is equal to eight. It is denoted as `blocked dgemm`.
- Sparse matrix-vector multiplication. The experiment is referred to as `spmv`.

The experiments were conducted on an extended version of the SESC simulator. SESC is a cycle-accurate execution-driven simulator built from MINT that emulates a MIPS processor [Renau et al., 2005]. Table 3.1 shows the parameters of the two-core CMP architecture. A large-size L2 cache results in very few capacity misses and nearly all cache misses are compulsory misses. Thus it is relatively easy to model large L2 caches. In order to study the more complicated non-compulsory misses, we choose to use a small L2 cache size.

3.4.1 Result for `dgemm`

Table 3.2 does a comparison between the actual number of misses and the predicted number of misses for running two threads. The relative error lies in the range between 1.97% and

Table 3.1: Parameters of the two-core simulated CMP architecture.

Processor	Two cores, 5.0GHz out of order issue
L1(private)	ICache: LRU, 4-way, 32KB 64B line, write-through DCache: LRU, 4-way, 8KB 64B line, write-through MESI protocol
L1L2 Bus	Split transaction system bus
L2 MSHR	64
L2(shared)	Unified, LRU, 64B line 64KB, fully associative write-back

20.19%. For each matrix size N, there are three rows that display the actual number of L2 cache misses when users run a single thread, the actual number when users run two threads, and the prediction for running two threads, respectively.

As shown in the third row for each N, the analytical model decomposes cache misses into three components: compulsory misses, capacity misses on private data, and capacity misses on shared data. Each component adopts a different approach to model. The compulsory and shared data misses are based upon empirical parameters, and the private data misses build upon a probability model. For different applications, the total number of cache misses is dominated by one of the three components. For instance, the `dgemm` experiment has a large number of capacity misses on shared data.

3.4.2 Result for `blocked dgemm`

This experiment is a tiled version of `dgemm`. It uses a block size of 8 to compute the matrix multiplication. Table 3.3 lists the actual number of misses for running one thread alone, the actual number for running two threads together, and the predicted number for running two threads. The relative error is between 0.1% and 4.2%.

3.4.3 Result for `spmv`

Finally, we conducted experiments on sparse matrix-vector multiplications. The matrices used are `dw2048` and `qc324` which were downloaded from the Matrix Market web site. `dw2048` is a 2048×2048 sparse matrix with 10114 non-zero elements, while `qc` is a 324×324 matrix having 26730 non-zero elements. Figures 3.5 and 3.6 show their images correspondingly. Table 3.4 lists the performance result. To estimate the number of compulsory misses for matrix `qc324`, we find that the two threads are working on nearly-disjoint subsets of the shared memory area, therefore we simply keep the number of compulsory misses unchanged.

Table 3.2: Result for `dgemm`: prediction of the total number of L2 misses for thread 0 if running with another thread. For each N, there are three rows. The 1st row shows the measured result for running a single thread, then the second row shows measured result for running two threads, and the third row shows our prediction.

N	Total	Compulsory	Capacity (private)	Capacity (shared)	Error
64 single	1041	1025	16		
64 dual	862	838	24		
64 predict	813	769	43	1	-5.68%
72 single	1313	1297	16		
72 dual	901	870	31		
72 predict	991	973	17	1	+9.99%
80 single	1631	1601	30		
80 dual	1096	1037	59		
80 predict	1233	1201	31	1	+12.50%
88 single	2076	1937	139		
88 dual	6479	1158	5321		
88 predict	7787	1453	145	6189	+20.19%
96 single	56839	2305	54534		
96 dual	30179	1681	28498		
96 predict	29584	1729	391	27464	-1.97%
104 single	72231	2705	69526		
104 dual	35531	2037	33494		
104 predict	37710	2029	1204	34477	+6.13%
112 single	89983	3137	86846		
112 dual	44428	2317	42111		
112 predict	46081	2353	607	43121	+3.72%
144 single	190445	5185	185260		
144 dual	93495	3817	89678		
144 predict	97135	3889	1229	92017	+3.89%
Average Error					8.01%

Table 3.3: Result for `blocked dgemm`: prediction of the total number of L2 misses for thread 0 if running with another thread. For each N, there are three rows. The 1st row shows the measured result for running a single thread, then the second row shows measured result for running two threads, and the third row shows our prediction.

N	Total	Compulsory	Capacity (private)	Capacity (shared)	Error
64 single	1047	1031	16		
64 dual	805	775	30		
64 predict	827	773	53	1	+2.73%
72 single	1251	1231	20		
72 dual	1952	985	977		
72 predict	1916	923	21	972	-1.84%
80 single	4537	1607	2930		
80 dual	2963	1215	1748		
80 predict	3089	1205	51	1833	+4.25%
88 single	5795	1855	3940		
88 dual	3374	1332	2042		
88 predict	3399	1391	71	1937	+0.74%
96 single	8161	2311	5850		
96 dual	4911	1764	3147		
96 predict	4792	1733	178	2881	-2.42%
104 single	9474	2607	6867		
104 dual	5319	1891	3428		
104 predict	5444	1955	108	3381	+2.35%
112 single	12690	3143	9547		
112 dual	7230	2423	4807		
112 predict	7202	2357	140	4705	-0.39%
144 single	26222	5191	21031		
144 dual	14543	3904	10639		
144 predict	14561	3893	299	10369	+0.12%
Average Error					1.85%

Table 3.4: Result for `spmv`: prediction of the total number of L2 misses for thread 0 if running with another thread. For each sparse matrix, the 1st row shows the measured result for running a single thread, then the 2nd row shows measured result for running two threads, and the 3rd row shows our prediction.

Matrix	Total	Compulsory	Capacity (private)	Capacity (shared)	Error
dw2048 single	1483	1403	80		
dw2048 dual	1483	1391	92		
dw2048 predict	1412	1324	88	0	-4.787%
qc324 single	2807	2693	114		
qc324 dual	2841	2668	173		
qc324 predict	2840	2693	147	0	-0.035%
Average Error					2.41%

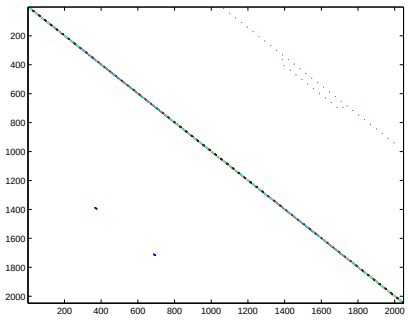


Figure 3.5: Sparse matrix of dw2048 ($nnz = 10,114$).

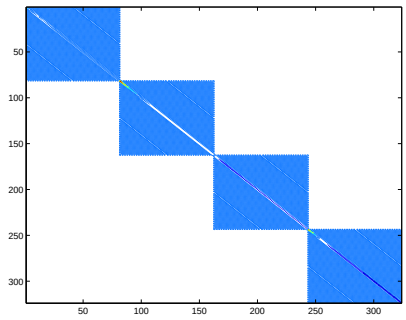


Figure 3.6: Sparse matrix of qc324 ($nnz = 26,730$).

3.5 Summary

This chapter presents an analytical model to predict the number of L2 cache misses on a chip multi-processor quantitatively. The model uses the circular sequence profiling and the stack processing technique to analyze an L2 cache trace. First, the trace file is scanned to generate a circular sequence profile. Next the analytical model takes as input the profile and outputs the number of cache misses for running multiple threads. Since we are concentrating on a fully associative L2 cache, cache misses are decomposed into three types: compulsory misses, capacity misses on shared data, and capacity misses on private data. Each miss type is modeled by a different method since each one's behavior is affected variously by other threads.

For all the three scientific programs, the model has an average relative error less than 8.01%. In addition, the analytical model provides an insight into how inter-thread cache sharing and cache contention affect program performance. Inspired by this insight, we start to investigate affinity based thread scheduling to optimize memory access efficiency for a more complex memory hierarchy. The following Chapters 4 and 5 describe the work of affinity based thread scheduling.

Chapter 4

Theoretical Framework for Affinity Thread Scheduling

4.1 Introduction

This chapter studies how to improve the memory effectiveness and maximize data reuse through affinity-based thread scheduling on multicore shared memory platforms. By using a large-scale DSM machine for a couple of years, we find that it is critical to improve user programs' memory access efficiency to speed up program performance. We run SGI's performance monitoring tool `pmshub` on the SGI Altix 3700 BX2 machine from NCSA. As an example, Fig. 4.1 shows that a number of user programs are experiencing a large amount of remote memory accesses on the DSM machine. The light yellow area indicates the number of remote memory accesses.

Therefore, we aim at searching for an optimal thread schedule to improve the memory effectiveness on all levels in the multi-level memory hierarchy (e.g., maximize cache reuse, reduce the number of remote memory accesses). Threads in our context refer to fine-grained user level threads that can be as small as a block of instructions for which a user program can create hundreds of thousands of such threads.

The material from this chapter was published in the 11th IEEE International Conference on Cluster Computing (IEEE Cluster 2009) [Song et al., 2009b].

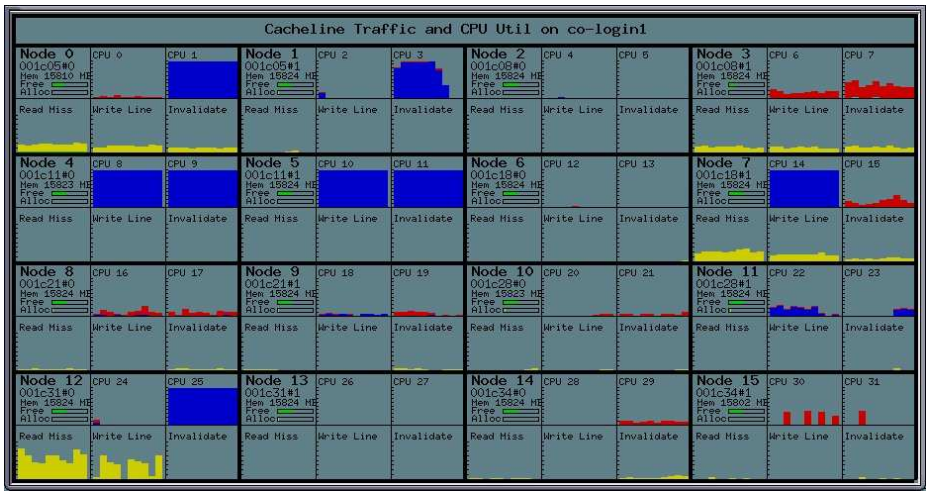


Figure 4.1: A screen shot of the performance monitoring tool `pmshub` on SGI Altix. The light yellow area reflects how many remote memory accesses have occurred.

To investigate the affinity-based thread scheduling problem, we introduce an analytical model to evaluate the cost of a thread schedule and then tackle it as an optimization problem. The analytical model consists of three submodels:

- *affinity graph submodel* for describing the affinity relationship between fine grain threads (or tasks) in a user program,
- *memory hierarchy submodel* for abstracting the memory hierarchy of a multicore system, and
- *cost submodel* for predicting the cost of a schedule to run the threads on the multicore system.

Our strategy is to let the affinity graph submodel characterize the user program and the memory hierarchy submodel characterize the system architecture. Then, in combination with the third cost submodel, we are able to answer the question "Given a multi-threaded program T and a shared-memory machine M, what is the cost to use a thread schedule A to execute program T on machine M?" Since finding an optimal thread schedule is NP-hard, we design a hierarchical graph partitioning algorithm to compute a near-optimal solution. Furthermore, an extension to support DAG scheduling is also proposed. The analytical

model is supported experimentally. We have applied the analytical model to two synthetic applications and a real-world application and showed that it can accurately measure the quality of a thread schedule.

4.2 The analytical model

4.2.1 The affinity scheduling problem

Given a set of single-application user-level threads $\{t_1, \ldots, t_m\}$ without data dependences, and a number of heterogeneous processors $p_1, \ldots, p_n$ located in a shared-memory hierarchy, find a good schedule A to achieve:

(a) maximal data reuse within a processor,

(b) minimal remote memory accesses, and

(c) load balancing.

Note that here we use threads to refer to user-level threads that can be as small as a block of instructions (e.g., task) or as large as a kernel-level thread (e.g., pthread).

Let schedule A be an onto function:

$$A : \{1, \ldots, m\} \longrightarrow \{1, \ldots, n\}, m \geq n.$$

$A(i) = j$ means put thread t_i on processor p_j. $A^{-1}(j)$ denotes the subset of threads running on processor p_j. We allow threads to have different workloads and processors to be heterogeneous with varying computational capabilities.

4.2.2 Affinity graph submodel

In order to decide on which processors to place two threads, it is critical to know whether there exist data items accessed in common by the two threads. We use the concept of "affinity" to quantify how many data items are accessed in common by a pair of threads.

Definition 4.1 (Affinity). *When two threads t_i, t_j access a number n of data items $\{x_1, x_2, \ldots, x_n\}$ in common, we say there is an affinity relationship between t_i and t_j, and affinity$(t_i, t_j) = n$ is the affinity strength.*

Because a user program has a set of threads, we introduce the concept of *affinity graph* to model the affinity relationship among the set of threads.

Definition 4.2 (Affinity Graph). *The affinity graph is an undirected weighted graph $G = \langle T, E, w_t, w_e \rangle$, where*

- $T = \{t_i$ *is a user-level thread* $\mid t_i$ *is data independent of* $t_j, \forall i \neq j\}$,
- E=$\{(t_i, t_j) \mid \exists$ *datum* x *such that both* t_i *and* t_j *access* $x\}$,
- $w_t : T \longrightarrow Z^+$ *denotes the amount of computation of each thread,*
- $w_e : E \longrightarrow Z^+$ *denotes the affinity strength between two threads. If* $(t_i, t_j) \notin E$, $w_e(t_i, t_j) = 0$.

Given an affinity graph $G = \langle T, E, w_t, w_e \rangle$, if $T_i \subset T$, we extend the definition of w_t and w_e to represent the weight of the subgraph for T_i. That is, $w_t(T_i) = \sum_{t \in T_i} w_t(t)$ and $w_e(T_i) = \sum_{t_i, t_j \in T_i} w_e(t_i, t_j)$.

Figure 4.2 shows an example of multiplying two 200×200 matrices. A and C are dense matrices. Matrix B has a special structure where the top right and bottom left blocks are zeros. We run four threads T0-T3 to compute the matrix multiplication. Threads T0-T3 compute the result for submatrices of C11, C12, C21, and C22 concurrently. The corresponding affinity graph is shown on the right hand side in Fig. 4.2.

We compare the performance of putting threads T0 and T2 on the same SMP node to that of putting T0 and T1 together (an intuitive way) for a system with two dual-CPU SMP nodes. Figure 4.3 shows the wallclock execution time of the two thread schedules. The optimized schedule is better than the original one by 20%. This example demonstrates that different thread schedules can result in significant performance difference.

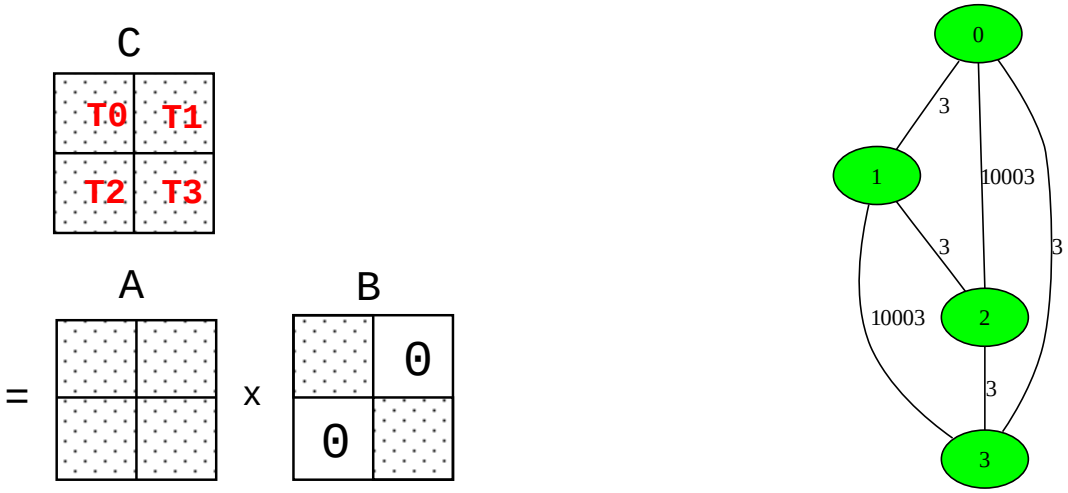


Figure 4.2: An example of affinity graph for matrix multiplication using four threads.

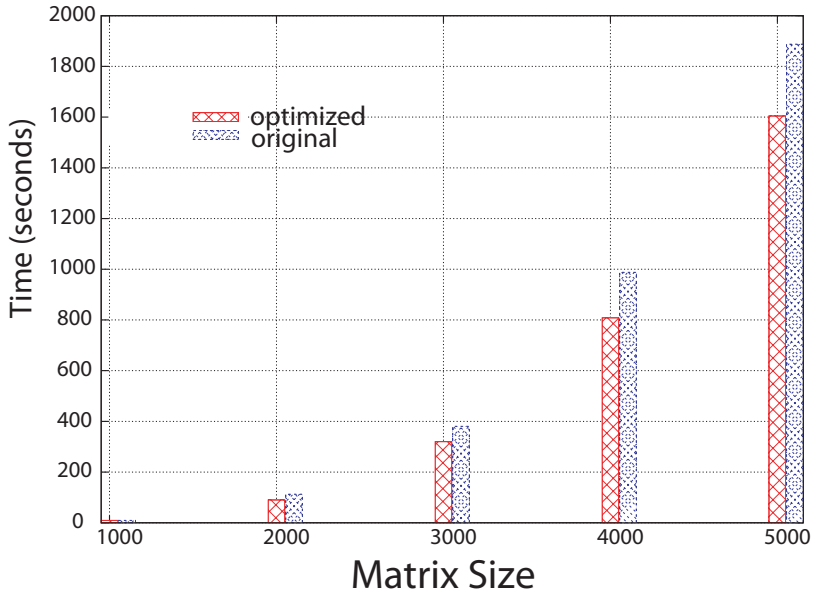


Figure 4.3: Comparison of two different thread schedules. The revealed affinity relationship (stronger affinity between T0 and T2) in Figure 4.2 helps improve the performance of the parallel matrix multiplication.

4.2.3 Memory hierarchy submodel

We assume that a shared-memory system has a hierarchical memory architecture. For instance, a number of processor cores may share an L2 or L3 cache. The data stored in the L2 and L3 caches are duplicated in the local main memory. Also the global address space includes the data stored in all the local memories. The hierarchical shared memory system is defined as follows:

Definition 4.3 (Memory Hierarchy Submodel)**.** *A shared-memory system R is a tree of the form $R = (r, T)$, where r is a memory node, T is the children of r and*

$$T = \{(r_i, T_{r_i}) \mid T_{r_i} \text{ is the children of } r_i\}.$$

The model assumes all the leaves are of the same height h (i.e., on the hth level), and all the edges on the same level l have identical weight w^l. Specifically, the leaf tree nodes $(r_i, \emptyset)$ denote processor cores and the interior tree nodes at levels $0 \ldots h-1$ denote memories. The model also assumes each memory contains a copy of the data in its children.

Figure 4.4 shows an example of a multicore DSM system. For convenience, we define the ancestor memories of a node n by $ancestor(n) = \{m :$ memory m resides on the path from the root to node n$\}$.

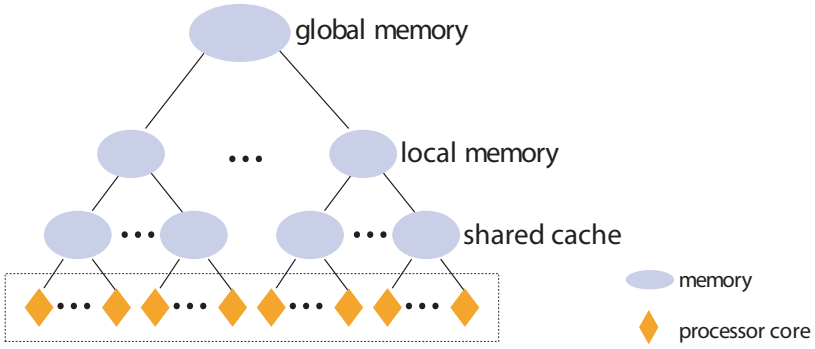


Figure 4.4: A 3-level memory hierarchy on a multicore DSM system.

Definition 4.4 (Memory Latency)**.** *If a processor p accesses datum x that is stored in memory m together with m's ancestor memories, we define memory latency $lat(p, x) = w(p, m)$, where*

$$w(p, m) = \sum_{edge\ e \in\ path\ p \to m} w^{level(e)}.$$

Lemma 4.1. *Let datum x reside in memory m together with m's ancestor memories, if processor p is a descendant of m but processor p' is not, then $lat(p, x) < lat(p', x)$.*

Proof. Let the lowest common ancestor of p and p' be $m2 = lca(p, p')$. Since p' is not under the subtree of m, $level(m2) < level(m)$. By Definition 4.4,

$$lat(p, x) = w(p, m) = w^{h-1} + w^{h-2} + \ldots + w^{level(m)},$$

$$lat(p', x) = w(p, m2) = w^{h-1} + w^{h-2} + \ldots + w^{level(m2)}.$$

Since $level(m2) < level(m)$, we know $lat(p', x) > lat(p, x)$. □

Corollary. *If two threads t_i and t_j access the same datum x stored in memory m, placing them on two processors located in the subtree of m minimizes $lat(t_i, x) + lat(t_j, x)$.*

Proof. By Lemma 4.1, placing thread t_i on a descendant processor p_i of m will minimize $lat(t_i, x)$. Similarly, $lat(p_j, x) = \min_{\forall p} lat(p, x)$ if p_j is another descendant processor of m. Therefore, placing the two threads on two processors in the subtree of m minimizes $lat(t_i, x) + lat(t_j, x)$ since both latencies are minimal. □

The corollary implies that the thread placement may affect the program performance if two threads have an affinity relationship.

4.2.4 Cost submodel

After identifying the affinity relationship between threads and the characteristics of the underlying architecture, we are now ready to estimate the cost of running a thread schedule.

Definition 4.5 (Cost Submodel)**.** *Given an affinity graph G, a shared-memory system M, and a thread schedule A, we define the cost to execute schedule A on system M as*

$$cost(G, M, A) = \sum_{\forall p_i, p_j} cost(A^{-1}(p_i), A^{-1}(p_j), M, G), \text{ where}$$

$$cost(T_i, T_j, M, G) = \sum_{t_i \in T_i, t_j \in T_j} w_e(t_i, t_j) lat(p_i, m_c), \text{ and}$$

m_c is the lowest common ancestor of processors p_i and p_j (i.e., $m_c = lca(p_i, p_j)$).

Lemma 4.2. *Assume a shared-memory system M has a set P of processors, and each memory m has k children such that m has k subsets $D(m)_i$ of processors (each child has a subtree and leads to a subset of processors), $i = 1 \dots k$. Suppose*

$$pair(m) = \{(p_x, p_y) \mid p_x \in D(m)_i, p_y \in D(m)_j, i, j \in [1, k]\},$$

then $\{pair(m) \mid m \in M\}$ is a partition for set $\mathcal{P} = P \times P \setminus \{(p_i, p_i) \mid p_i \in P\}$.

Proof. We need to show (1) $\bigcup_{m \in M} pair(m) = \mathcal{P}$, and (2) $pair(m_i) \cap pair(m_j) = \emptyset$.

(1.1) To prove $\bigcup_{m \in M} pair(m) \subseteq \mathcal{P}$.

Let $\{p_x, p_y\} \in pair(m_i)$ for a certain $m_i \in M$. By definition of $pair(m_i)$, we know $p_x \in D(m_i)_{k_1}$ and $p_y \in D(m_i)_{k_2}$. Sine M is a tree, $p_x \neq p_y$. Therefore, $(p_x, p_y) \in \mathcal{P}$, such that $\bigcup_{m \in M} pair(m) \subseteq \mathcal{P}$.

(1.2) To prove $\mathcal{P} \subseteq \bigcup_{m \in M} pair(m)$.

Let $\{p_x, p_y\} \in \mathcal{P}$ and m_c be the lowest common ancestor of p_x and p_y. Assume m_c has k subsets $D(m_c)_i$ of processors that are derived from its k children (or branches), then p_x and p_y must belong to different branches of m_c (since otherwise m_c won't be the lowest common ancestor). WLOG, let $p_x \in D(m_c)_{k_1}$ and $p_y \in D(m_c)_{k_2}$, $k_1 \neq k_2$. Therefore, $(p_x, p_y) \in pair(m_c)$, such that $\mathcal{P} \subseteq \bigcup_{m \in M} pair(m)$.

(2) To prove $pair(m_i) \cap pair(m_j) = \emptyset$ if $m_i \neq m_j$.

Suppose $\exists (p_x, p_y) \in pair(m_i) \cap pair(m_j)$ and $m_i \neq m_j$. By definition of $pair$, m_i is the lowest common ancestor of p_x and p_y. Similarly, m_j is also the lowest common ancestor of

p_x and p_y. Since p_x and p_y determine a unique lowest common ancestor, thus $m_i = m_j$, contradicting the assumption of $m_i \neq m_j$. □

Theorem 4.1. *Suppose a schedule A runs a set of threads in affinity graph G on a system M. Let $time_l$ denote $lat(p, p$'s ancestor memory at level $l)$ and M_l denote the set of memories at level l, then $cost(G, M, A)$ can also be expressed as:*

$$\sum_{l=0}^{h-1} \sum_{m \in M_l} \sum_{\substack{(p_i,p_j) \\ \in pair(m)}} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) \times time_l.$$

In other words,

$$cost(G, M, A) = \sum_{l=0}^{h-1} time_l \times SharingOnLevel_l,$$

where $SharingOnLevel_l$ denotes the amount of affinity between threads that access those memories located on level l. That is,

$$SharingOnLevel_l = \sum_{m \in M_l} \sum_{\substack{(p_i,p_j) \\ \in pair(m)}} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y).$$

Proof. Suppose processors p_i and p_j have the lowest common ancestor memory $lca(p_i, p_j)$. By Definition 4.5,

$$cost(G, M, A) = \sum_{p_i \neq p_j} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) lat(p_i, lca(p_i, p_j))$$

$$= \sum_{(p_i,p_j) \in \mathcal{P}} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) time_{level(lca(p_i,p_j))}.$$

By Lemma 4.2,

$$\sum_{(p_i,p_j) \in \mathcal{P}} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) time_{level(lca(p_i,p_j))}$$

$$= (\sum_{m \in M} \sum_{(p_i,p_j) \in pair(m)}) \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) time_{level(m)}.$$

Since every memory $m \in M_l$ for a certain l,

$$cost(G, M, A) = ((\sum_{l=0}^{h-1} \sum_{m \in M_l}) \sum_{\substack{(p_i,p_j) \\ \in pair(m)}}) \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) time_{level(m)}.$$

By $time_{level(m)} \in \{time_0, \ldots, time_{h-1}\}$, and all memories $\in M_l$ have the same $time_l$,

$$cost(G, M, A) = time_0 \sum_{m \in M_0} \sum_{(p_i,p_j) \in pair(m)} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y)$$

$$+time_1 \sum_{m \in M_1} \sum_{(p_i,p_j) \in pair(m)} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) + \cdots$$

$$+time_{h-1} \sum_{m \in M_{h-1}} \sum_{(p_i,p_j) \in pair(m)} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y).$$

That is,

$$cost(G, M, A) = \sum_{i=0}^{h-1} time_i \times SharingOnLevel_i.$$

□

4.3 The optimization problem

Given an affinity graph $G = \langle T, E, w_t, w_e \rangle$ and a shared-memory system M, the problem of finding an optimal schedule A^* such that $cost(G, M, A^*) = \min_{\forall A} cost(G, M, A)$ can be considered as an integer linear programming problem.

4.3.1 An integer linear programming problem

Suppose $time_i > time_{i+1} > 0$ and M is of height h. Let

$$x_i = SharingOnLevel_i = \sum_{m \in M_i} \sum_{(p_x,p_y) \in pair(m)} \sum_{\substack{t_x \in A^{-1}(p_x) \\ t_y \in A^{-1}(p_y)}} w_e(t_x, t_y)$$

denote the sum of affinity strength on level i ($0 \leq i \leq h-1$), and $x_h = \sum_{p_i} \sum_{t_x,t_y \in A^{-1}(p_i)} w_e(t_x, t_y)$ denote the sum of affinity strength within each processor. The ILP problem is formulated as follows:

1. Minimize $\sum_{i=0}^{h-1} x_i \times time_i$
2. Subject to

   $x_0 + x_1 + \ldots + x_{h-1} + x_h = w(E)$,

   $x_i \in Z^+$ for $i \in [0, h-1]$, and

   $\{x_0, x_1, \ldots, x_{h-1}\}$ is derived from a load balanced thread schedule that distributes the set of threads T across n processors evenly.

Note that the values of x_i's are also constrained by load-balanced thread schedules.

By the following Lemma 4.3, the classic graph partitioning problem is reducible to the problem of minimizing $cost(G, M, A)$ if M's edges have the same weight that is in turn reducible to the problem of minimizing $cost(G, M, A)$ for arbitrary M. Since the classic graph partitioning problem is NP-hard, finding an optimal schedule to minimize $cost(G, M, A)$ is also NP-hard.

Lemma 4.3. *Given a graph G and a shared-memory machine M with n processors. If $time_0 = time_1 = \ldots = time_{h-1}$ on M, P^* is an optimal n-way classic graph partitioning for G iff A^* is an optimal thread schedule for M, where P^* and A^* are of the same family of subsets.*

Proof. First we need to show: if P^* is an optimal n-way classic graph partitioning, then A^* is an optimal thread schedule. Suppose $P^* = \{T_1, \ldots, T_n\}$ is an optimal n-way partition of

graph G and $time_0 = \ldots = time_{h-1} = c$, then

$$\sum_{T_i,T_j \in P^*} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c = \min_{\forall P} \sum_{T_i,T_j \in P} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c.$$

Since schedule A^* has the same partition as P^* such that $A^*(T_i) = p_j$ for some p_j,

$$\sum_{T_i,T_j \in P^*} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c = \sum_{p_i,p_j} \sum_{\substack{u \in {A^*}^{-1}(p_i) \\ v \in {A^*}^{-1}(p_j)}} w_e(u,v)c.$$

$$\text{Thus, } \sum_{p_i,p_j} \sum_{\substack{u \in {A^*}^{-1}(p_i) \\ v \in {A^*}^{-1}(p_j)}} w_e(u,v)c = \min_{\forall P} \sum_{T_i,T_j \in P} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c.$$

Since $\forall$ partition P,

$$\sum_{T_i,T_j \in P} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c = \sum_{p_i,p_j} \sum_{\substack{u \in A^{-1}(p_i) \\ v \in A^{-1}(p_j)}} w_e(u,v)c,$$

$$\sum_{p_i,p_j} \sum_{\substack{u \in {A^*}^{-1}(p_i) \\ v \in {A^*}^{-1}(p_j)}} w_e(u,v)c = \min_{\forall A} \sum_{p_i,p_j} \sum_{\substack{u \in A^{-1}(p_i) \\ v \in A^{-1}(p_j)}} w_e(u,v)c.$$

That is, A^* is an optimal thread schedule for M if A^* and P^* define the same family of subsets. The converse can be proved in a similar manner. □

4.3.2 An approximation algorithm

Similar to *cut* in the classic graph partitioning problem, we use *share* to express the affinity strength between two partitions:

$$share(T_x, T_y) = \sum_{\forall u \in T_x, \forall v \in T_y} w_e(u,v),$$

where T_x and T_y are two disjoint thread sets.

If processor cores on a system have different computational powers, we use a *partition distribution vector* to define Unbalanced Graph Partitioning. Given affinity graph $G = \langle T, E, w_t, w_e \rangle$ and $W = w_t(T)$, the partition distribution vector $\langle d_1, d_2, \ldots, d_n \rangle$ defines a partition $\{P_i\}$ whose weight $w_t(P_i) = d_i \times W$ and $\sum_i d_i = 1$. A more powerful processor core will be assigned a larger portion of the computational tasks accordingly. The graph partitioning algorithm uses the partition distribution vector to guarantee that the workload on each core is load balanced. Our implementation calls the METIS function `METIS_WPartGraphKway` to perform the unbalanced partitioning.

There are two optimization goals in our partitioning process: conforming to the partition distribution vector and minimizing the sharing between partitions. We propose a greedy hierarchical partitioning algorithm to divide the affinity graph according to the partition distribution vector. The goal is to minimize the sharing between partitions in the order of level 0 to level $h-1$ in a top-down fashion. Assume a parallel system has n compute nodes, each of which has p processors. (a) We divide the threads into n sets $N_1, N_2, \ldots, N_n$ by minimizing

$$\sum_{1 \le i,j \le n} share(N_i, N_j), i \neq j.$$

Now each N_i has been assigned a set of threads. Since each compute node also has p processors, (b) then partition each N_i into p sets $P_1, P_2, \ldots, P_p$ by minimizing the sharing between two processors:

$$\sum_{1 \le i,j \le p} share(P_i, P_j), i \neq j.$$

Theorem 4.2. *Let n be the number of processors, G be a graph, M be a system of height h. Assume P^* is an optimal n-way balanced graph partitioning, then a hierarchical graph partitioning algorithm can find a $(2, n)$-way graph partition P such that*

$$\frac{cost(P)}{cost(P^*)} \le h, \text{ where } cost(P) = \sum_{T_i, T_j \in P} share(T_i, T_j).$$

Proof. The proof adopts similar scheme used by [Simon and Teng, 1997]. Basically we want to show that the total cost happening at each level is at most C^*.

Suppose $P^* = \{G_1, G_2, \ldots, G_n\}$ with $C^* = cost(P^*)$ and $B = |G_i|$. At the root level of the partitioning tree, it is possible to just group G_i's into k subsets of equal size by only removing the inter-block (i.e., inter-G_i) edges. An alternative is to use the optimal k-way partitioning to find a partitioning whose total cost is smaller than that of grouping G_i's, where $G_i \in P^*$. Thus, the total cost at the root level is at most C^*.

Those G_i's may be partitioned into pieces after the top i-1 levels of partitioning. Suppose a node at level i-1 consists of a subset of the pieces from G_i's. Similar to the proof for bounding the cost at the root level, the cost to partition each node of level i-1 is smaller than that of removing only the inter-block edges. Thus, at each level of partitioning, the total cost between the subgraphs is no more than C^*.

$$\text{Therefore, } cost(P) \leq hC^* \Longrightarrow \frac{cost(P)}{cost(P^*)} \leq h \ .$$

Each node performs an optimal k-way partitioning operation for different k's. Suppose a node at level i contains a number s of G_i's pieces. The k-way partitioning of the node N results in a subgraph size at most $\frac{sB}{k} + B$ (by bin-packing theorem). Then the k-way partitioning is of

$$\rho = (\frac{sB}{k} + B)/\frac{sB}{k} = (1 + k/s) <= 2.$$

□

Theorem 4.3. *Suppose an optimal thread schedule A^* has $cost(G, M, A^*)$. Then the hierarchical graph partitioning algorithm can find a schedule A such that*

$$\frac{cost(G, M, A)}{cost(G, M, A^*)} \leq h\frac{time_0}{time_{h-1}}.$$

Proof. By the definition of $cost(G, M, A)$,

$$\frac{cost(G, M, A)}{cos(G, M, A^*)} = \frac{\sum_{i=0}^{h-1} SharingOnLevel_i \times time_i}{\sum_{i=0}^{h-1} SharingOnLevel_i^* \times time_i} \leq \frac{time_0 \sum_{i=0}^{h-1} SharingOnLevel_i}{time_{h-1} \sum_{i=0}^{h-1} SharingOnLevel_i^*}.$$

$$\text{Since } cost(P) = \sum_{i=0}^{h-1} SharingOnLevel_i \text{ if } A^{-1} \text{ has the same partition as } P,$$

$$\text{by Theorem 4.2, } \frac{cost(G, M, A)}{cos(G, M, A^*)} \leq h \frac{time_0}{time_{h-1}}.$$

□

4.4 Extension to support DAG scheduling

The previous hierarchical partitioning algorithm assumes that threads have no data dependence. When threads are dependent on each other and hence form a DAG, we need to extend the algorithm to deal with DAG scheduling. In this extension, given a DAG G, we divide G into a number of levels (horizontally), each of which consists of a subset of independent threads. This step can be achieved by analyzing G and determining the longest path from the root to each node. The total number of levels is equal to the length of the critical path. Within each level, we use the the hierarchical graph partitioning algorithm to determine a good schedule to run the threads. Due to data dependences, no thread in level $i + 1$ can start until all threads in level i complete. We call this simple approach "greedy multi-level thread scheduling". Figure 4.5 depicts how to divide a DAG into four levels.

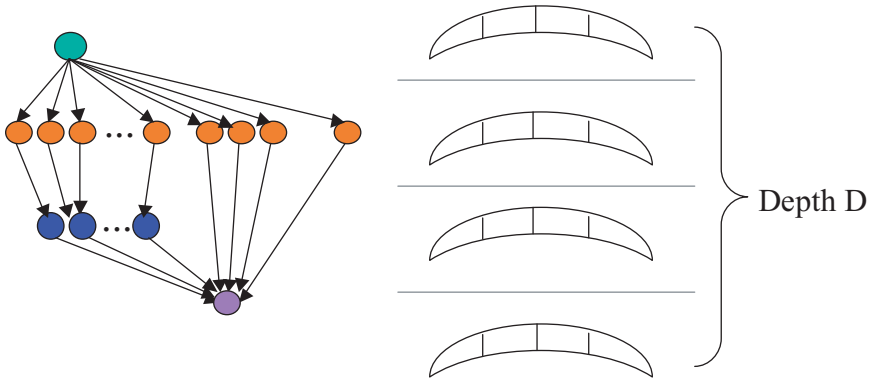


Figure 4.5: An example of greedy multi-level thread scheduling. The DAG is divided into four levels. The level index of each node is equal to the length of the longest path from the root to that node.

Lemma 4.4. *Suppose a DAG has D levels and the total amount of computation is W. If each thread t_i computes an amount $w(t_i)$ of work, where $w(t_i) \in [0,1]$, then the greedy multi-level thread scheduler for p processors takes at most $\frac{W-D}{p} + D$ time.*

Proof. Let s_i denote the amount of work on level i, where $i \in [1, D]$.

$$Time = \sum_{i=1}^{D} \lceil \frac{s_i}{p} \rceil \leq \sum_{i=1}^{D} (\frac{s_i}{p} + (1 - \frac{1}{p})) = \frac{W}{p} + D - \frac{D}{p} = \frac{W-D}{p} + D.$$

□

Theorem 4.4. *The greedy multi-level thread scheduling method has an approximation ratio of $1 + \frac{D}{(\frac{W}{p})}$.*

Proof. Let C and C^* represent the actual execution time and the optimal execution time, respectively. It is easy to show that any execution time is at least $\max(W/p, T_\infty)$.

$$\text{By } C \leq \frac{W-D}{p} + D \text{ and } C^* \geq \frac{W}{p},$$

$$\frac{C}{C^*} \leq \frac{(W-D)/p + D}{C^*} \leq \frac{(W-D)/p + D}{(\frac{W}{p})} = 1 + \frac{(1-1/p)D}{(\frac{W}{p})} < 1 + \frac{D}{(\frac{W}{p})}.$$

□

Based on Theorem 4.4, if $W/D > p$, then the greedy multi-level scheduling method takes time at most twice the optimal time. In addition, programs with fine-grain threads often satisfy the condition of $W/D > p$, and are commonly found in scientific applications (e.g., Cholesky, LU, QR factorizations).

4.5 Evaluation of the analytical model

This section describes how to evaluate the analytical model and presents the analysis of two scientific experiments to which we have applied the optimized thread schedules. The optimized thread schedules are computed by the hierarchical partitioning algorithm.

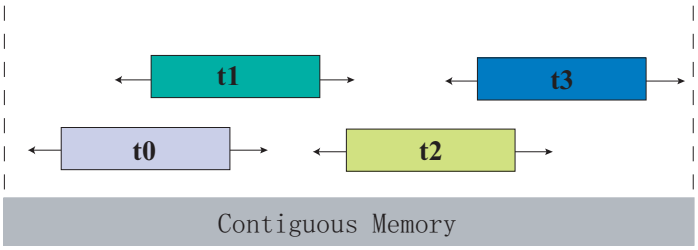


Figure 4.6: Four threads access a contiguous memory of size 128MB. Each thread occupies 32MB and their positions are arbitrary.

We conducted three experiments on a distributed shared-memory SGI Altix machine that has two compute nodes, each of which has two processors. In the experiments, four threads are executing on four processors. In terms of complexity, the experiments range from simple, synthetic to real-world applications.

The first two synthetic experiments allocate a contiguous memory block of size 128M bytes. Each thread accesses 1/4 of the 128MB memory. Each thread first initializes its memory segment with some values and then computes the sum of the square of each element. The position of a memory segment could be anywhere as long as it is within the range of the 128MB memory. The affinity strength between two threads is equal to the size of the overlapping area between their footprints. Figure 4.6 illustrates how four threads access a contiguous block of 128MB memory.

The first experiment is the simplest one where the memory segments of the four threads are disjoint (i.e., evenly distributed and affinity=0). Then we gradually move thread $t1$ towards thread $t0$ so that the overlapping area of $t0$ and $t1$ becomes bigger and bigger. Since there is no affinity change between the four threads except for the pair of $t0$ and $t1$, we only compare two thread placements: placing $t0$ and $t1$ together on the same node (i.e., `(t0,t1)(t2,t3)`), and placing them separately on two nodes (i.e., `(t0,t2)(t1,t3)`).

Figure 4.7 shows that the actual performance (i.e., execution time in seconds) of the placement `(t0,t2)(t1,t3)` becomes worse and worse with the increment of the overlapping footprint. The cost estimated by the analytical model has the same trend as the actual performance. Note that the cost model is not intended to predict the execution time, but

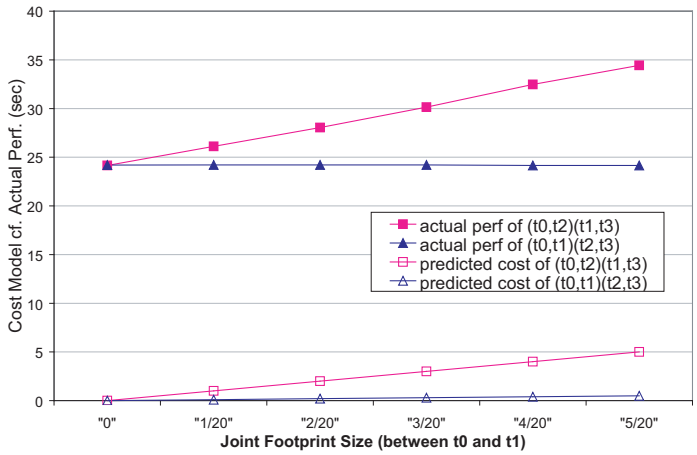


Figure 4.7: Compare the predicted cost to the actual execution time for two schedules: `(t0,t1)(t2,t3)` and `(t0,t2)(t1,t3)`.

used to measure the quality of a thread schedule, and a ranking is sufficient to find the best thread schedule.

The second experiment performs ten program runs each of which has a different memory footprint pattern. All the footprint patterns are generated randomly. The pattern generation uses a random number generator to create a starting position $addr_i$ for each thread t_i such that t_i accesses addresses in the range of $[addr_i, addr_i + 32MB)$. Given 4 threads on two SMP nodes each with 2 processors, there are totally $\binom{4}{2}/2! = 3$ thread schedules. We denote them as `(t0, t1)`, `(t0, t2)`, and `(t0,t3)`, respectively. For each of the ten footprint patterns, the same program runs three times each with one of the three thread schedule. The experiment also computes the predicted cost for each run and compares it to the actual measurement.

Figure 4.8 demonstrates that the cost of the three schedules consistently reflects the ranking of their actual program performance. In other words, given any footprint pattern, if schedule A has a cost bigger than schedule B, the actual performance of the program using schedule A is slower than that using schedule B.

Furthermore, we apply the analytical model to an important kernel in many scientific applications: sparse matrix vector product (`SpMV`). The sparse matrices were downloaded from the UF Sparse Matrix Collection [Davis, 1997]. For many matrices, an optimized

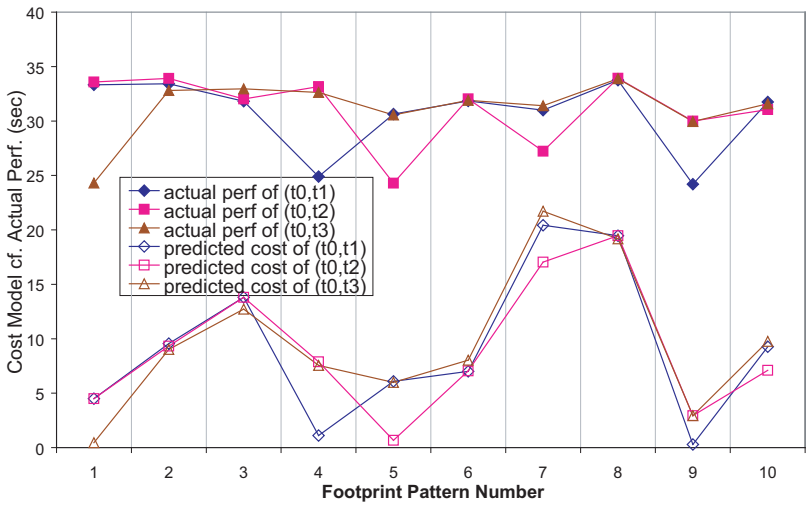


Figure 4.8: Compare the predicted cost to the actual execution time for three thread schedules on 10 randomly generated memory footprint patterns.

thread schedule can improve the program performance by 15% to 25%. But for a few other matrices, the program using the optimized thread schedule instead runs 1% slower than the original program.

We use the analytical model to analyze this phenomenon and investigate the differences between the old and the new thread schedules. We pick two cases from our experiments conducted on a distributed shared memory machine. One experiment multiplies the sparse matrix `msc01440` and improves the performance by 23%. The other one multiplies the sparse matrix `circuit_1` but is slower than the original program by 1%. Tables 4.1 and 4.2 list the values of $SharingOnLevel_i$ (defined in Theorem 4.1) for the original and new schedules, respectively. Assuming the remote memory access time is $time_0$ and the local memory access time is $time_1$, we can compute $cost(G, M, A)$ by adding $time_0 \times SharingOnLevel_0$ and $time_1 \times ShareingOnLevel_1$ easily.

As shown in Table 4.1, the new schedule reduces the remote memory access cost by 68.5% and thus improves program performance. However, based on Table 4.2, there is only a small reduction by using the new schedule (4.4% in remote memory accesses). Due to the overhead of executing the new schedule, the actual performance shows a 1% slowdown instead of a small speedup.

Table 4.1: Apply the analytical model to study why `SpMV` is improved with input `msc01440`.

Affinity on different levels	Original schedule	New schedule	Reduction
Remote memory	54,175	17,043	68.5%
Local memory	107,765	13,964	87.0%

Table 4.2: Apply the analytical model to study why `SpMV` is not improved with input `circuit_1`.

Affinity on different levels	Original schedule	New schedule	Reduction
Remote memory	127,620	121,973	4.4%
Local memory	230,759	206,076	10.7%

4.6 Summary

While there exist tools and runtime systems to schedule threads efficiently, little is known about what would be an optimal affinity thread schedule to maximize the memory effectiveness and why it is optimal. This chapter introduces an analytical model to evaluate the performance of a thread schedule. The model has three submodels: an affinity graph submodel to describe the affinity relationship between threads, a memory hierarchy submodel to characterize the underlying shared-memory architecture, and a cost submodel to estimate the cost for a certain thread schedule. The experimental results show that the analytical model can accurately estimate the cost of a thread schedule for two synthetic programs and a real-world application.

The chapter also proposes a hierarchical graph partitioning algorithm to determine near-optimal solutions based on which we have designed and implemented a Feedback Directed Optimization (FDO) framework. Next chapter describes the FDO framework.

Chapter 5

Feedback Directed Affinity Thread Scheduling

5.1 Introduction

This chapter introduces a feedback directed framework to improve the memory access effectiveness by identifying user-level threads and reorganizing them to enhance the temporal and spatial locality, as well as placing tightly-coupled threads as close as possible. The user-level thread may be considered as a "logical task" whose granularity varies from as tiny as a single instruction to as big as an actual kernel-level thread. With the help of data dependence analysis, it is even possible to reorder individual instructions to maximize data reuse across the entire data set. However for scalability and quick analysis, we identify fine-grained user-level threads as conceptual units of computation from the viewpoint of a programmer. Most of the time it is straightforward to identify a thread. For instance, an iteration of a loop nest or an update of an object may be created as a thread.

To improve the data locality, we need to decide which user-level threads should be in the same group and in what order to execute them, as well as how to map the groups to different processors. Figure 5.1 illustrates the overall structure of our approach to realizing

The material from this chapter was published in the 16th International Symposium on High-Performance Distributed Computing (HPDC 2007) [Song et al., 2007a].

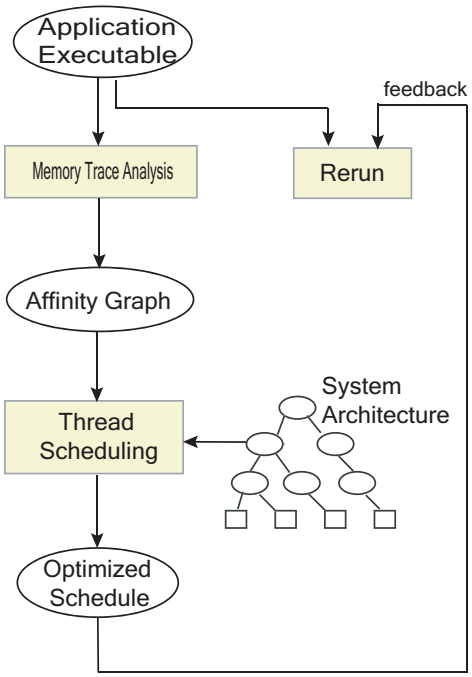


Figure 5.1: Overall structure of the thread scheduling method with memory considerations.

it. The approach relies upon a binary instrumentation tool to (i) obtain and analyze the memory trace of each thread and find out the nature of memory sharing between threads in an "affinity graph". As defined in Sect. 4.2.2, an affinity graph is an undirected weighted graph where each vertex represents a thread and the weight for edge e_{ij} denotes the total number of distinct addresses accessed in common by threads i and j. To make the memory tracing method more practical, the instrumentation tool generates affinity graphs dynamically without storing the huge memory trace to disk. After the instrumented executable finishes, an affinity graph is built and written to a file. Next, (ii) we partition the graph into a number of subgraphs (in our experiments, the number is equal to the number of processors). Based on the partitions, (iii) we compute a "good" schedule to put threads on different processors correspondingly. The schedule is written in a file which will be later used as feedback to future executions. Finally, (iv) a user reruns the program taking as input the feedback file.

We have experimented with the feedback-directed thread scheduling method using numerous application programs, including sparse matrix-vector multiplication (stored in compressed row storage or compressed column storage format), sparse matrix-matrix multiplication, a kernel from computational fluid dynamics codes, and Cholesky factorization. The experimental results show that the feedback-directed method can significantly reduce the execution time. In particular, the method is able to improve performance for programs with dynamic memory access patterns and little compile-time information.

5.2 Feedback directed method

A feedback-directed method strives to improve performance by using profiling information to exploit opportunities for optimizations. Our work is concerned with how to maximize data locality on each processor and minimize the data sharing between processors (each processor runs one kernel thread). Since we determine a thread schedule based on the referenced addresses, collecting a memory trace is typically necessary. There are four types of approaches to obtaining the memory trace: compiler-based, run-time system, online feedback-directed optimization (FDO), and offline feedback-directed optimization. We choose to use the offline FDO method due to the following reasons:

- The feedback-directed method can attain dynamic information about memory references, particularly for programs with irregular access patterns. The dynamic information cannot be obtained at compile time.
- The offline method has much less overhead than online methods since the optimized thread schedule is determined during the very beginning profiling program run.
- Our method collects a more accurate and detailed memory trace for each thread to determine an optimized thread schedule. Current run-time systems usually use a couple of heuristics to schedule threads.

5.3 Memory trace analysis

We design a memory trace analysis tool to collect the memory trace of user programs. The analysis tool supports binary code instrumentation and is built upon Pin [Luk et al., 2005]. Pin follows the model of ATOM and allows tool writers to analyze applications at the instruction level. It uses a dynamic just-in-time (JIT) compiler to instrument binary codes while they are running. The set of Pin APIs provide support for observing a process's architectural states (e.g., register contents and memory references).

We write a memory trace analysis tool in C++ and use the Pin API to implement two types of routines: *instrumentation routine* and *analysis routine*. The instrumentation routine tells Pin to insert instrumentation to every instruction that reads or writes data. The virtual address of the referenced datum is passed as an argument to the analysis routine. In order to differentiate addresses from distinct threads, the `thread ID` is passed as another argument to the analysis routine. Instrumenting the binary executable also enables us to keep the original memory access pattern while reflecting various compiler optimizations.

`RecordMemRead` and `RecordMemWrite` are the two corresponding analysis routines for read and write operations. The analysis routine works as an event handler. For each memory reference, the routine identifies the thread ID for that reference and stores it into a buffer. Each thread owns a buffer for keeping distinct addresses (i.e., two references to the same address will be stored once). We tried writing the memory trace to a file, but the size of the trace file and the I/O cost increased so fast that it soon became impractical to use.

The space requirement of our memory trace analysis tool is equal to the sum of the distinct addresses referenced by each thread. It is possible for us to impose a limit on the number of distinct addresses for each thread so that all the tracing operations can be performed in memory (i.e., without disk). This diskless tracing method is much less expensive and more practical than writing the memory trace to disk.

5.4 Techniques to process large affinity graphs

While a modern computer system has no problem keeping the distinct addresses in memory for most applications, the size of the generated affinity graph can explode very quickly. For instance, 10^5 fine-grained threads (10^5 vertices) may require 40GB memory if the graph is totally connected (10^{10} edges). The size of the graph is limited by the capacity of the memory. Given a fixed amount of memory, it is not trivial to build an appropriately sized affinity graph without exceeding the available memory.

We adopt several techniques to reduce the time complexity and space complexity of the process of graph creation. It rarely happens in practice that every thread has a memory affinity relationship to all the other threads if we do not consider the small number of global variables. Thus affinity graphs are often sparse and symmetric. We represent affinity graphs by an adjacency list and only store edges e_{ij} where i < j to reduce the memory requirement. This is analogous to storing an upper-triangular adjacency matrix.

A simple algorithm to build an affinity graph from the recorded memory trace is shown in Fig. 5.2. The memory trace analysis tool instruments and executes the executable. After executing the program, the tool stores addresses in `thrd_addrs` for each thread. Next, the algorithm compares the memory trace of every two threads i and j where i < j. The function `create_edge()` creates an edge between thread i and thread j and assigns a proper weight to the edge.

Suppose there are T threads and each thread accesses N addresses. Even though the function `create_edge()` has a linear time complexity $O(N)$ (by merging two ordered lists),

```
1 map<tid, addr> thrd_addrs;
2 for i = 1, num_thrds-1
3   for j = i+1, num_thrds
      /*Compare traces of threads i, j.*/
4     create_edge(thrd_addrs, i, j);
5   end for
6 end for
```

Figure 5.2: A simple algorithm to build affinity graphs.

the overall time complexity is equal to $O(T^2N)$. In practice N is often bounded but T could be very large (e.g., 10^6 to 10^8). Furthermore, no matter how sparse a graph is, this algorithm always takes time $O(T^2N)$ to build the graph, which could lead to many hours of computation.

To be more efficient in processing large graphs, we change to a different data structure and develop a new algorithm. The new algorithm employs an adjustable parameter of $DenseRatio \in [0 \dots 1]$ to control the density of the graph. If an address is accessed by all the threads, the graph will become fully connected. However this address will not help the next step of graph partitioning. Therefore we adjust *DenseRatio* to eliminate those edges that form a clique of size greater than $NumberThreads \times DenseRatio$. $DenseRatio = 0$ yields a graph with no edges, while $DenseRatio = 1$ yields a graph with all possible edges.

Figure 5.3 lists the pseudo-code for the more efficient algorithm. The new data structure `addr_thrds` stores *address* as keys instead of prior *thread_id* as keys. The new algorithm takes as input the memory trace stored in `addr_thrds` and builds an affinity graph. Given an address, the algorithm determines the number of threads that have accessed it. If the number is greater than the total number of threads $\times$ *DenseRatio*, we skip both the trace analysis for the address and the creation of relevant edges. Otherwise, we create edges among the set of threads.

Again, let N be the total number of addresses referenced by the application and T be the total number of threads. The time complexity of the algorithm varies between $O(N)$ and $O(NT^2)$ depending on how sparse the graph is. The worst-case time complexity occurs when the graph is fully connected and $DenseRatio = 1.0$.

Certainly we can use *DenseRatio* to reduce the graph density. Note that the previous algorithm in Fig. 5.2 always has a time complexity of $O(NT^2)$. The new algorithm not only has a better average-case time complexity but also uses a tunable parameter of *DenseRatio* to eliminate particular edges. In practice, we use $DenseRatio = 0.9$ to reduce the number of edges and are able to create sparse graphs. In addition, most of the eliminated edges are due to a couple of global variables. Note that a single global variable can lead to a fully connected graph.

Since we adopt a diskless approach to analyzing the memory trace, the ability to record traces is limited by the amount of memory. To reduce the memory requirement, We collect a partial memory trace for each thread. An alternative would be to to represent contiguous memory addresses by blocks instead of great numbers of individual addresses.

Furthermore, executing the complete instrumented executable could be very time consuming (e.g., 10 to 100 times) than the original executable due to the cost of calling the analysis subroutine, storing the memory trace in associated arrays, and creating graphs in the end. So we run a partial execution of the truncated program to overcome the problem. For instance, a single outer-loop iteration of an iterative method is sufficient to build an affinity graph.

The generated affinity graphs can be written either in Graphviz DOT format for the purpose of displaying, or in Chaco or METIS format for the next stage of graph partitioning. We implement the hierarchical algorithm described in Sect. 4.3.2 to partition the generated affinity graphs to determine optimized thread schedules.

5.5 Applications

We performed experiments with a variety of applications to evaluate our feedback-directed method of thread scheduling. These applications cover a number of domains and have been widely used by users.

```
1 T = total number of threads;
2 map<addr, tid> addr_thrds;
3 for each addr in addr_thrds do
4   thrd_set = threads accessing addr;
5   m = size of thrd_set;
6   if (m > DenseRatio*T) continue;
7   create edges between any pair of threads within thrd_set;
8 end for
```

Figure 5.3: A more efficient algorithm to build affinity graphs.

```
1 for iter = 1, NUM_ITER
2   #pragma omp parallel for
3   for i = 1, num_rows
4     y = 0;
5     for j = row[i], row[i+1]-1
6       y += val[j]*x[col[j]];
7     end for
8     y[i] = y;
9   end for
10end for
```

Figure 5.4: Parallel iterative method calling SpMV $y = Ax$. Sparse matrix A is stored in CRS format by arrays `val, col, row`.

5.5.1 Sparse matrix-vector multiplication

Sparse matrix-vector multiplication (SpMV) is an important subroutine in many iterative methods. We use two formats of Compressed Column Storage (CCS) and Compressed Row Storage (CRS) to implement iterative methods and try to improve their performance. The program using the CRS format is presented in Fig. 5.4. The inner `for` loop is distributed to a number of processors and executed in parallel. Arrays `val, col, row` store the sparse matrix `A` while `x, y` store the column vectors for computing $y = Ax$. The code using the CCS format is similar to that in Figure 5.4 and not shown here.

We use the technique of affinity graph partitioning to improve program performance for both CRS and CCS formats. Suppose matrix A is stored in CCS format, the vector $y = Ax$ is computed by $y = \sum_{j=1}^{n} a_j x_j$, where a_j is the jth column of A. For the parallel version with p threads, each thread computes a partial sum and updates the vector y. But an update of element y_i may invalidate a set of neighboring y_i' elements in other processors due to false sharing. By grouping the columns that access the same elements in vector y, we are able to improve the temporal locality and reduce the chances of false sharing on vector y.

As for sparse matrix A stored in CRS format, each thread computes a subset of vector y, where $y_i = \sum_{j=1}^{n} a_{ij} x_j$. Note that the only chance of data reuse lies in vector x. If two

rows read the same set of elements $x_{s1}, x_{s2}, \ldots, x_{s_d}$ in vector x, running them continuously will reuse the d elements and improve the temporal locality.

5.5.2 Sparse matrix-matrix multiplication

In sparse matrix-matrix multiplication (SpMM), we store matrix A row by row in CRS format and store matrix B column by column in CCS format. The parallel SpMM program is described in Fig. 5.5. Sparse matrices are distributed along rows across a number p of processors. Each processor computes for a number of $\frac{N}{p}$ consecutive rows in matrix C.

The outer two loops `i` and `j` in Fig. 5.5 are data independent and can be executed in parallel. A user-level thread may either compute a dot product of the ith row of matrix A and the jth column of matrix B (i.e., the outer two loops are parallelized), or multiply the ith row of A and the whole matrix of B (i.e., only the outermost loop `i` is parallelized). The former definition is more fine-grained and can identify a greater number of threads to reorder to maximize the program locality. But it is too costly to compute since the number of threads N^2 may result in a very large graph. Instead, we use the latter coarse-grained definition to do our experiments. The experimental results demonstrate that we can still speedup the program by up to 20% even using the coarse-grained threads.

5.5.3 Computational fluid dynamics kernel

The computational fluid dynamics (CFD) kernel implements an iterative irregular-mesh partial differential equation (PDE) solver abstracted from computational fluid dynamics applications [Pingali et al., 2003]. The irregular meshes are used to model physical structures and consist of nodes and edges. The computational kernel iterates over the edges of the mesh, computing the forces between both end points of each edge. It then modifies the values of all nodes. Figure 5.6 shows the parallel version of the CFD kernel.

Each edge corresponds to a user-level thread in our experiment. We partition the threads into p sets (for p processors) to maximize data reuse by grouping together the threads accessing the same nodes. In addition, for each processor, we reorder the threads in the processor's subgraph to maximize cache reuse by means of the breadth-first traversal.

```
1 struct CRS A;
2 struct CCS B;
3 double    *C;
4 #pragma omp parallel for
5 for i = 1, N
6   for j = 1, N
7     c = C[i*N+j];
8     for idx_a = A.row[i], A.row[i+1]-1
9     for idx_b = B.col[j], B.col[j+1]-1
10      if(A.col[idx_a] == B.row[idx_b])
11        c += A.val[idx_a] * B.val[idx_b];
12    end for
13    end for
14    C[i*N+j] = c;
15  end for
16end for
```

Figure 5.5: Parallel version of SpMM.

```
1 for iter = 1, NUM_ITER
2   #pragma omp parallel for
3   for i = 1, edges
4     n1 = left[i];
5     n2 = right[i];
6     force = f(x[n1],x[n2]);
7     y[n1] += force;
8     y[n2] -= force;
9   end for
10end for
```

Figure 5.6: Parallel version of the CFD kernel.

5.5.4 Cholesky factorization

Given an $n \times n$ symmetric positive definite matrix A, Cholesky factorization computes $A = LL^T$ where L is an $n \times n$ lower triangular matrix. For efficiency, we use the right-looking blocked algorithm so that we can apply Level-3 BLAS directly to a block of matrix A. The blocked Cholesky factorization algorithm works as follows:

$$\text{Given } A = \begin{pmatrix} A_{1:b,1:b} & A_{1:b,b+1,n} \\ A_{b+1:n,1:b} & A_{b+1:n,b+1:n} \end{pmatrix},$$

$$\text{compute } L = \begin{pmatrix} L_{1:b,1:b} & 0 \\ L_{b+1:n,1:b} & L_{b+1:n,b+1:n} \end{pmatrix}$$

by calling:

(i) Level-3 BLAS `POTRF` to solve $L_{1:b,1:b}$,

$$A_{1:b,1:b} = L_{1:b,1:b}L^T_{1:b,1:b}$$

(ii) Level-3 BLAS `TRSM` to solve a linear equation system for $L_{b+1:n,1:b}$ in parallel,

$$L_{b+1:n,1:b}L^T_{1:b,1:b} = A_{b+1:n,1:b}$$

(iii) Level-3 BLAS `GEMM` to compute a rank-r update on the trailing $A_{b+1:n,b+1:n}$ in parallel,

$$A'_{b+1:n,b+1:n} \leftarrow A_{b+1:n,b+1:n} - L_{b+1:n,1:b}L^T_{b+1:n,1:b}.$$

We apply the above 3 steps repeatedly to $A'_{b+1:n,b+1:n}$ until A' consists of a single block.

The program is shown in Fig. 5.7. Variable `A_ij` refers to a block which is located in the ith row and jth column in terms of blocks. Given an $n \times n$ matrix and a block of size b, `nblocks` $= n/b$. To use the greedy multi-level algorithm described in Section 4.4, we must know the program's task graph. Figure 5.8 shows the corresponding task graph for a 4 block by 4 block matrix and its level division. The figure displays only one iteration of

```
1 for k = 1, nblocks
2   dpotf2(A_kk);
3   #pragma omp parallel for
4   for j = k+1, nblocks
5     dtrsm(A_kk, A_jk);
6   end for
7   for i = k+1, nblocks
8     #pragma omp parallel for
9     for j = k+1, i
10      dgemm(A_ik, A_jk, A_ij);
11    end for
12  end for
13end for
```

Figure 5.7: Parallel tiled Cholesky factorization.

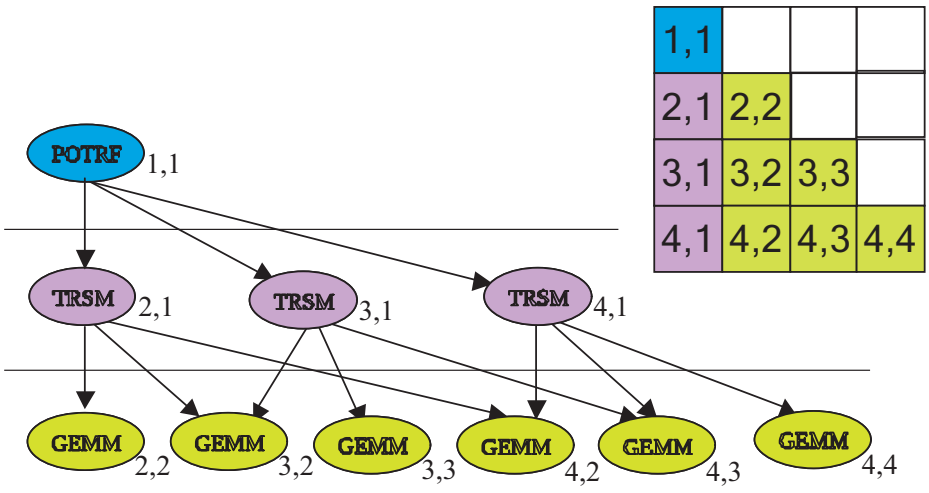


Figure 5.8: Cholesky factorization DAG (one iteration).

the outer loop. The other iterations have a similar structure and are not shown here. Each task in the DAG corresponds to a Level-3 BLAS operation.

5.6 Application measurement results

We conducted experiments on two platforms. One platform is a multicore SMP machine consisting of two sockets, each of which has a quad-core 2.66 GHZ Intel Clovertown chip. Since the set of two cores on each chip share an L2 cache, the corresponding memory hierarchy has two levels: the main memory on the machine and the L2 caches on each chip.

The other platform is an SGI Altix 3700 BX2 system with 256 compute nodes. Each node has two 1.6 GHZ Intel Itanium processors. The system has a ccNUMA Distributed Shared Memory (DSM) that is physically distributed across different nodes. Every processor can access any memory location through the SGI NUMAlink 4 interconnect. The memory access time depends on the distance between the processor and the node where the physical memory is located. Its corresponding memory hierarchy also has two levels: the virtual global memory and memories on each compute node.

For each program, we always compare the performance of the program using the optimized thread schedule to that of the program built by compiler optimizations.

5.6.1 Commands to run experiments

There are 3 steps to run experiments with optimized thread schedules.

1. Use the trace analysis tool to execute a user executable. The output is an affinity graph. Note that the affinity graph is reusable on a different system.

   ```
   pin -mt -t mem_tracer -- <prog> <args>
   ```

2. Use the hierarchical partitioning tool to compute an optimized thread schedule based on the target machine. The output is the new thread schedule.

   ```
   partition_graph <prog>.graph <machine>.sys
   ```

3. Run the user executable again using the new thread schedule.

```
prog <args> <schedule>
```

5.6.2 Implementation issues

We implement all the parallel applications in C using Pthreads. We obtain an optimized thread schedule automatically through our memory trace analysis tool described in Section 5.2. In order to execute the program with the new schedule, we need to make a minor change to the original program manually. This step could be done easily by extending a compiler. For instance, instead of running `for i = 0 to n`, the compiler can wrap the fundamental computation unit by `for i = mytasks[0] to mytasks[n]`, where `mytasks` is specified in the new thread schedule.

After finding the affinity relationship between threads, we use the `cpuset` library provided by SGI Linux [SGI, 2006] to bind Pthreads to different processors. Note that the binding of Pthreads to processors happens only once when they are created. The cpuset API (e.g., `cpuset_pin()` and `cpuset_membind()`) allows our C programs to place both processor and physical memory within a cpuset.

5.6.3 SpMV

Sparse matrix-vector multiplication is called by a synchronous iterative method. For each sparse matrix, we run a fixed number of iterations. There are two versions written for SpMV. One version uses the CCS storage format and the other one uses the CRS format. Users often choose one of the two formats to implement their programs. Table 5.1 lists all the sparse matrices used in our SpMV experiments. They were downloaded from the UF Sparse Matrix Collection [Davis, 1997]. A matrix is chosen if the amount of computation for each row is approximately equal.

Figure 5.9 shows the effect of the optimization on the SGI Altix machine. Values less than 1 indicate performance speedup. Our scheduling method reduces execution times by 10% to 25% for four out of seven matrices in both CCS and CRS formats. The improvement for the CRS program comes from maximized data reuse, and the improvement for the

Table 5.1: Sparse matrices used in the SpMV experiment.

	Name	Dimension	NNZ
1	msc01440	1,440	44,998
2	circuit_1	2,624	35,823
3	bp_1000	822	4,661
4	coater1	1,348	19,457
5	msc23052	23,052	1,142,686
6	lhr01	1,477	18,592
7	utm3060	3,060	42,211

CCS program comes from reduced false sharing. Only three matrices (circuit_1, lhr01, and coater1 in CCS format) show a little slowdown ($< 5\%$).

On the Intel Clovertown machine (Fig. 5.10), only two matrices (circuit_1 in the CCS format and utm3060 in the CRS format) are not improved. But the average performance improvement on the Intel Clovertown machine is not as good as that on the SGI Altix machine. The reason is because a cache miss on the SGI machine may result in a remote memory access, while a miss on the Intel machine results in a much cheaper local memory access. Therefore, the new thread schedule could save more time on the SGI machine than on the Intel machine through reducing the number of cache misses.

5.6.4 SpMM

We compute sparse matrix-matrix multiplication $C = AB$ with four test inputs which are shown in Table 5.2. Instead of attempting various combinations of pairing sparse matrices, we simply let B equal the transpose of A. In the experiment, matrix A is stored row by row in the CRS format and matrix B is stored by columns in the CCS format. We let a user-level thread compute the product of the ith row of A and the whole matrix of B. Figure 5.11 shows that the performance improvement on the SGI machine is about 20% on average, and Fig. 5.12 shows that the improvement on the Intel machine is between 1% and 12%.

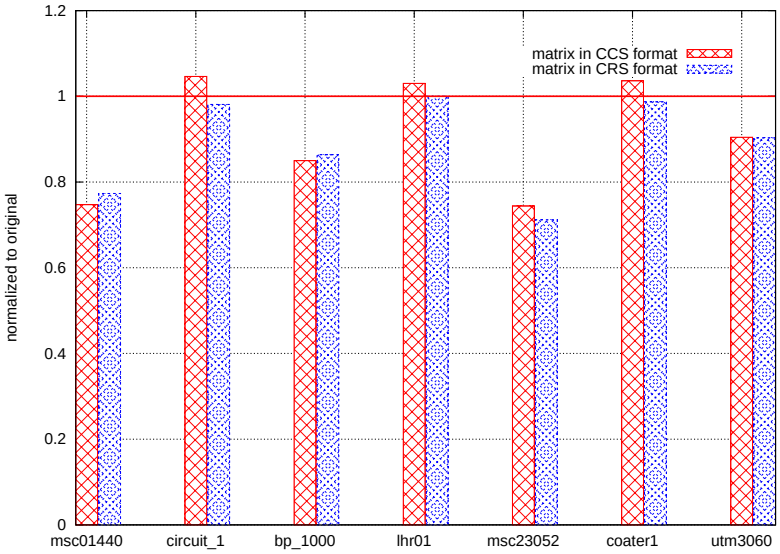


Figure 5.9: SpMV on SGI Altix with 4 processors. Both formats of CCS and CRS are used for each sparse matrix.

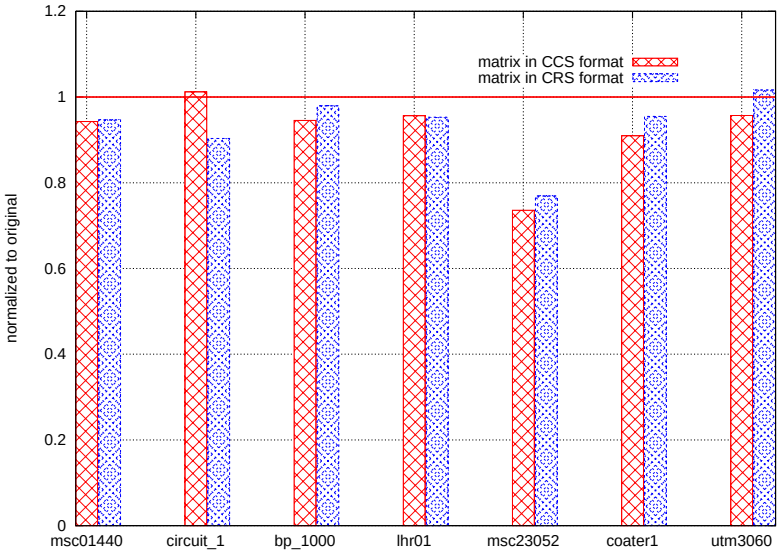


Figure 5.10: SpMV on Intel Clovertown with 8 cores. Both formats of CCS and CRS are used for each sparse matrix.

Table 5.2: Sparse matrices used in the SpMM experiment.

	Name	Dimension	NNZ
1	msc01440	1,440	44,998
2	msc01050	1,050	26,198
3	utm3060	3,060	42,211
4	bcsstk13	2,003	83,883

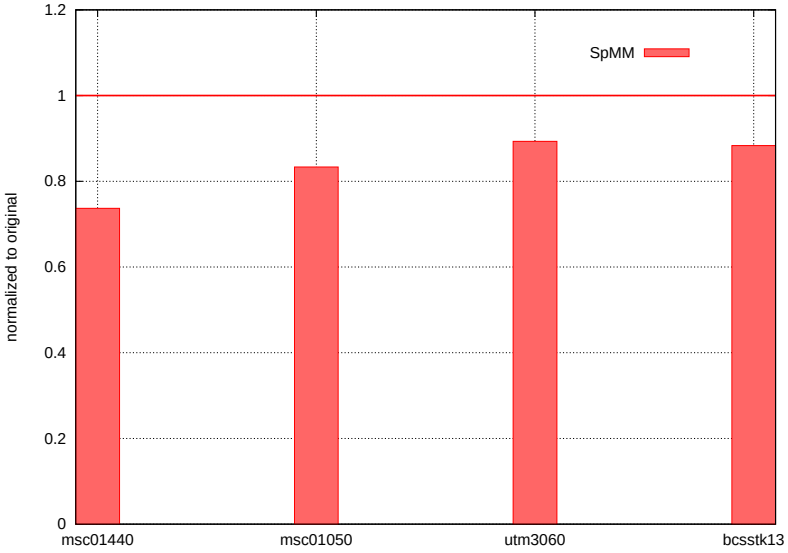


Figure 5.11: SpMM on SGI Altix with 4 processors.

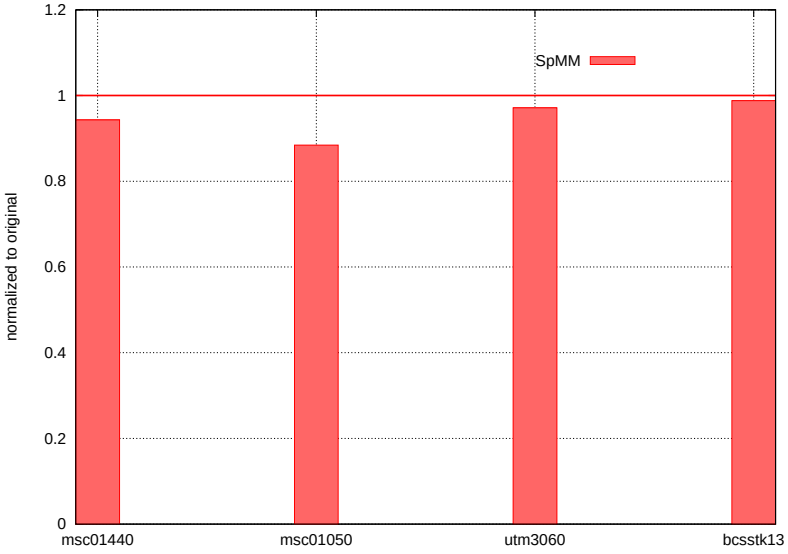


Figure 5.12: SpMM on Intel Clovertown with 8 cores.

5.6.5 CFD kernel

On the Intel Clovertown SMP machine, we compare the program using the optimized thread schedule to the original program over a number of irregular meshes. In the experiments, a mesh always has 10 times more edges than vertices. Figure 5.13 shows that using the optimized thread schedule reduces the execution time by 25% to 35% on the Intel SMP machine.

On the SGI Altix machine, we vary the number of processors to compare the program performance. The CFD kernel always takes as input a mesh of $40,000$ vertices and $400,000$ edges. For different numbers of processors (e.g., 4, 8, 16, and 32), using the new thread schedule reduces the execution time by 32% to 42%, as depicted in Fig. 5.14. The performance improvement on the SGI machine is larger than that on the Intel machine since the SGI machine has a distributed shared memory architecture.

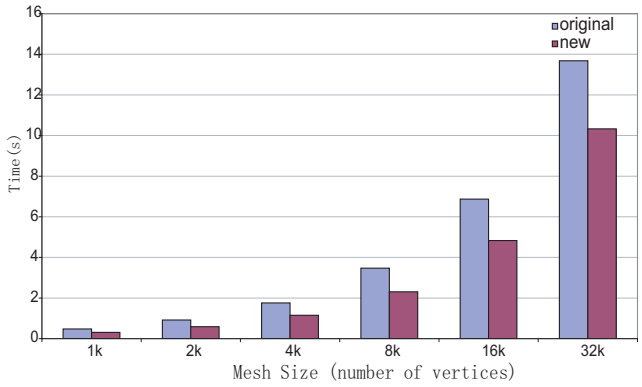


Figure 5.13: CFD kernel on Intel Clovertown with 8 cores.

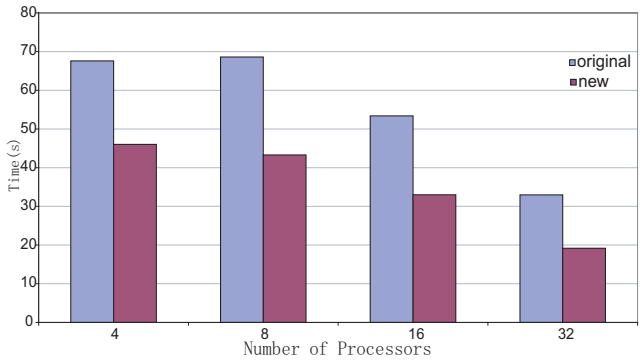


Figure 5.14: CFD kernel on SGI Altix.

5.6.6 Cholesky factorization

Unlike the previous experiments with independent threads, Cholesky factorization has threads with data dependences. We use the greedy multi-level thread scheduling method to determine an optimized thread schedule for the corresponding DAG level by level. Compared to the original schedule that allocates threads to processors in a block distribution way, the new schedule improves not only data locality but also load balance. The related numerical results have been verified.

For comparison, Fig. 5.15 also displays the performance of Intel MKL 9.1. On the Intel Clovertown machine, we can see that the new program is 60% to 200% faster than the original one, while the MKL library always provides better performance than the original one.

On the SGI machine, we conducted experiments using different numbers of processors (4, 8, 16) and compared the performance with that of MKL 7.2. Each experiment takes as input matrices with different sizes. Figures 5.16, 5.17, and 5.18 demonstrate that the new program is faster than the original program by 30% to 400% for different number of processors. The speedup on SGI Altix is greater than that on Intel Clovertown is because the optimized schedule on Altix not only reduces the number of cache misses, but also reduces the number of remote memory accesses due to the NUMA architecture.

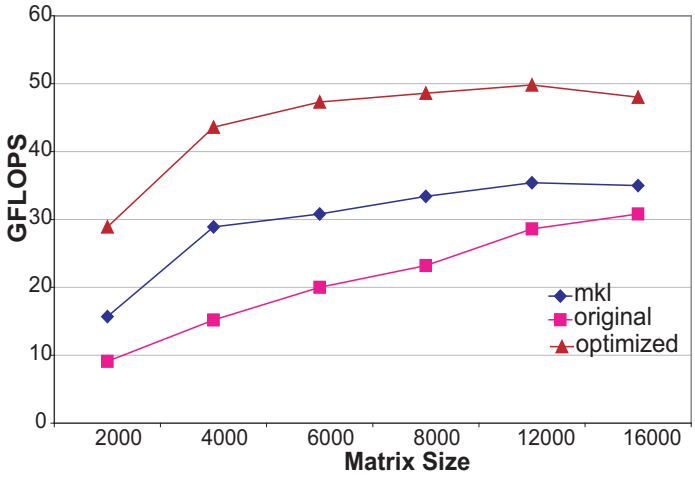


Figure 5.15: Cholesky factorization on Intel Clovertown.

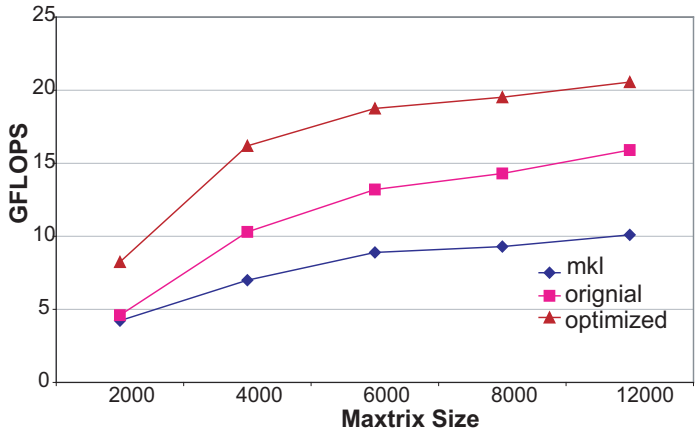


Figure 5.16: Cholesky factorization on SGI Altix with 4 processors.

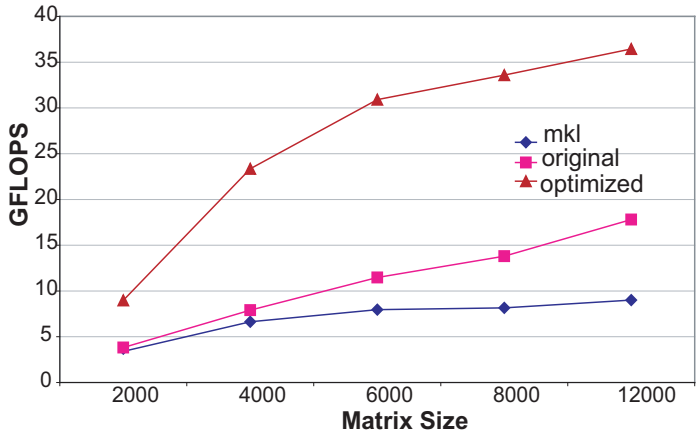


Figure 5.17: Cholesky factorization on SGI Altix with 8 processors.

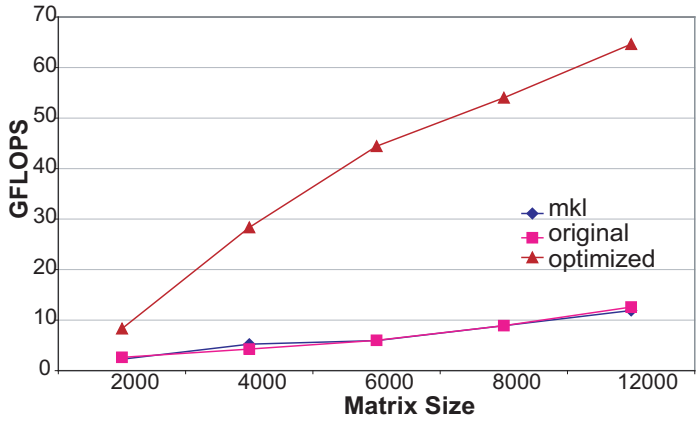


Figure 5.18: Cholesky factorization on SGI Altix with 16 processors.

5.7 Summary

We have designed and implemented a feedback-directed optimization framework to maximize program locality on shared-memory systems. We first run a binary instrumentation tool to automatically identify the nature of memory sharing between threads which is represented by an affinity graph. Next we perform graph partitioning to determine an optimized thread schedule to assign threads to different processors. The optimized thread schedule improves the data locality of the original program and reduces TLB misses as well as the number of remote memory references on a DSM system.

Experiments on four different types of applications have shown that our method is effective and can reduce the program execution time greatly. Although we are using a trace-based method, the online trace analysis completely eliminates expensive disk I/O operations. We demonstrate that the feedback-directed method is particularly good for applications with irregular computation and dynamic memory access patterns. The overhead to execute an application using the optimized thread schedule is also small. It would be possible to implement the feedback-directed framework in compilers so that the programmer can achieve high-performance on shared memory systems automatically.

Chapter 6

Dynamic DAG Scheduling

6.1 Introduction

Given a processor with hundreds or even thousands of cores, it is critical to increase the degree of thread-level parallelism to utilize all the available cores to improve program performance [Asanovic et al., 2006] [Buttari et al., 2009]. Our goal is to create as many concurrent tasks as possible to prevent processor cores from becoming idle. The number of synchronization points such as supersteps must be minimized as well since a potentially large number of tasks could be ready to execute but are stalled within the following supersteps.

We strive to design linear algebra software that is going to scale well on both shared-memory and distributed-memory multicore systems. Our approach to developing the scalable software is to put fine-grained computational tasks in a directed acyclic graph (DAG) and schedule them dynamically. To achieve high scalability, we propose a decentralized scheduling scheme for distributed-memory systems. That is, each node runs a private runtime system and communicates with other nodes regarding data dependences only when necessary. The runtime system has no globally shared data structures, no requirement of large storage space for DAGs, and no blocking operations. Furthermore, it respects critical

The material from this chapter was published in the 22nd ACM/IEEE Supercomputing Conference (SC'09) [Song et al., 2009c].

paths and keeps load balance. The runtime system is composed of three types of threads: task-generation thread, computing thread, and communication thread. At any time, only a small portion of the graph is stored in memory. The task-generation thread generates tasks sequentially and stores tasks in a fixed-size *task window* (i.e., building vertices of the graph). The computing thread analyzes the relationship between the tasks in the task window and solves data dependences automatically (i.e., building edges of the graph). Although the data dependences are solved in a distributed manner, we prove that the scheduling mechanism is dead-lock free even with a small-size task window and out-of-order message delivery.

The linear algebra programs are data-availability driven. The execution of a DAG starts from the entry task of the graph and finishes with the exit task. Whenever all the inputs of a task are available, the task becomes ready and will be executed by an idle processor. For easy use, our linear algebra software uses the same interface as `ScaLAPACK`. While offering scalability guarantee, the dynamic DAG scheduling mechanism is transparent to users. Instead of implementing every linear algebra algorithm from scratch, we wrote a task-based library to generate tasks for the basic subroutines (e.g., PBLAS) so that a new algorithm is simply a combination of a few task-based subroutines.

We apply the runtime system to a class of dense linear algebra algorithms: Cholesky factorization, LU factorization, and QR factorization. The factorization algorithms we used are tiled algorithms [Buttari et al., 2009]. In the tiled algorithms, each task computes an LAPACK or a Level-3 BLAS subroutine. The tiled algorithms can fully utilize the Level-3 BLAS operations such that the cache hit rate is maximized and data movement is minimized. We conducted experiments on both shared-memory and distributed-memory machines. The experimental results demonstrate that the distributed task scheduling approach is efficient and scalable.

6.2 Task-based linear algebra library and programs

Most LAPACK and ScaLAPACK algorithms are composed of a small number of of fundamental operations [Blackford et al., 1996] [Choi et al., 1996]. The fundamental operations

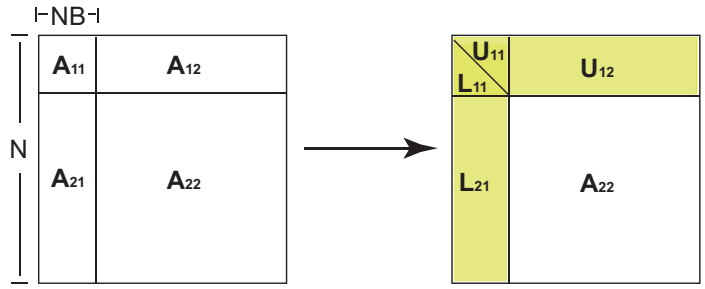


Figure 6.1: Block LU factorization.

are implemented as Level-2 or Level-3 BLAS routines. For instance, the Cholesky, LU, and QR factorizations all repetitively perform two operations: panel factorization and trailing submatrix update. The panel factorization transforms the leftmost collection of columns (i.e., column panel) followed by the trailing submatrix update using the panel factorization result. For instance, Fig. 6.1 shows an example of block LU factorization. First, the $N \times NB$ panel is factorized by the LU factorization. Next, after some pivoting, we solve the block row U_{12} using L_{11}. Finally, we update the submatrix A_{22} by multiplying the previously computed L_{21} and U_{12}.

Each fundamental operation can be regarded as a black box. We wish to keep the same interface but execute the program asynchronously in order to eliminate unnecessary barriers between operations (i.e., data-flow driven). To minimize the programming effort, we propose to use a task-based library to replace the fundamental subroutines. We let the new task-based subroutine generate a set of tasks and let a runtime system execute them dynamically. To understand how it works, Fig. 6.2 shows the pseudocode that implements the LU factorization with our task-based library. The subroutines of `PDGETF2_T`, `PDTRSM_T`, `PDGEMM_T` simply generate tasks and put them into the task window which is a member of the runtime system data structure `RTS_context`. One can build new linear algebra algorithms directly upon the task-based library.

Programs written with the task-based library will be executed by a single thread called *task-generation thread*. The task-generation thread executes the serial task-based program and creates tasks one by one to keep the original sequential semantics. The number of tasks to be generated are constrained by the size of the task window. Whenever an empty window

```
for (k=0; k < nblks; k++) {
  /* panel factorization */
  PDGETF2_T(nblks-k, 1, A, k, k, &RTS_context);
  /* compute block row of U */
  PDTRSM_T(1, nblks-k, A, k, k, &RTS_context);
  /* trailing submatrix update */
  PDGEMM_T(nblks-k, 1, A, k, k, 1, nblks-k, A, k, k,
           nblks-k, nblks-k, A, k, k, &RTS_context);
  /* deallocate panel blocks */
  for (i = k; i < nblks; i++)
    Release_Block(A, i, k, &RTS_context);
}
```

Figure 6.2: Block LU factorization program written with the task-based subroutines.

slot is available, the task-generation thread will start and create a new task. A finished task will be removed from the task window immediately. Any task in the task window will eventually become ready and be executed by a computing thread after all of its parents are completed. Since the task-generation thread does not do any computation, it takes a very small percentage of the CPU time. The execution of a task-based program is started by the task-generation thread placing the entry task of the DAG into the task window. An idle computing thread picks up the entry task and fires new tasks after finishing it.

6.3 Distributed data dependence solving

It is not trivial to generate tasks and solve data dependences in a distributed environment without much communication. Processes running on different nodes execute the same program and generate the same set of tasks so that a single task corresponds to a number of instances (each process has one). A correct algorithm requires all the processes make a uniform decision with respect to which consumer task to fire and how to make sure the consumer task is fired only once. Other complex issues include which process should execute a specific task and how to handle tasks with multiple outputs but belonging to different processes. This section first introduces a centralized algorithm, then describes how to use block data layout and various tasks modes to extend it to a distributed algorithm.

6.3.1 A centralized version

On a shared-memory machine, a single task-generation thread executes the user program sequentially and maintains the serial semantic order between tasks. We use a single linked task list to maintain the task order. If there exists a data dependence, the task list can determine which task precedes another. Figure 6.3 illustrates how to detect a RAW (read after write), WAR (write after read), or WAW (write after write) dependence based on the task list. A task always takes a number of inputs and writes to one or more outputs. Therefore, tasks stored in the task list keep information such as the input and output memory locations. Whenever two tasks access the same memory location and one of them is write, the runtime system detects a data dependence and stalls the successor till the predecessor is finished. Since WAR and WAW dependences can be removed by renaming, we only consider the true dependence (RAW) in our algorithm.

There are two operations to access the task list: *APPEND* and *FIRE*. The task-generation thread generates a new task t_j and invokes the APPEND operation to put task t_j to the end of the task list. Before the appending, APPEND scans the task list from the list head to check if there exists a task t_i such that t_i writes to datum x and t_j reads datum x (i.e., a data dependence). If none of the previous tasks writes to task t_j's kth

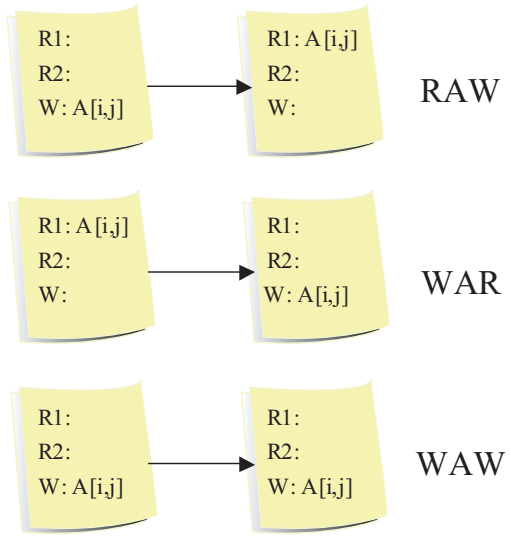


Figure 6.3: Detecting data dependences based on a task list.

input, we set the status of t_j's kth input as "ready". When all of t_j's inputs are ready, task t_j becomes a ready task.

After a task completes and modifies its output datum y, the FIRE operation starts to search for the tasks that read the datum y. Instead of from the head of the list, the runtime system scans the task list from the position of the completed task to the end of the list to find which tasks are waiting for y. The scanning process exits after confronting the first task that writes to y. We denote the set of scanned tasks that are linked between the completed task and the exit point as S. If a task is in S and one of its inputs is of datum y, that input will be set as "ready". Since we track data dependences for data blocks and use a fixed-size task window, the time and space overhead is not expensive. Section 6.4.3 discusses the space overhead of the method.

6.3.2 Block data layout and task assignment

The technique of block data layout is used to improve memory hierarchy performance [Park et al., 2003]. In the block data layout, a matrix is divided into submatrices of size NB $\times$ NB. Data elements within a block are stored contiguously in memory. On a distributed memory system, we use the 2D cyclic distribution method to map matrix blocks to compute nodes. Assuming a process grid of $P_r \times P_c$, a matrix block $A[I, J]$ will be mapped to process $[I \ mod \ P_r, J \ mod \ P_c]$. Note that we always map a block as an indivisible unit. We bind each task to its output data such that the computation is centered around data to minimize data movement and maximize data locality. Suppose the output of a task t is $A[I, J]$, then we assign t to process $[I \ mod \ P_r, J \ mod \ P_c]$. Since data blocks are allocated by the 2D cyclic method statically, the tasks assignment to each process is also static. That is, if the output block of a task is mapped to process p_i, then p_i owns that task correspondingly. But the task scheduling within a process is dynamic.

In addition to reducing the communication cost, the 2D cyclic distribution of blocks and tasks has a few more advantages. It has been proven to have the following important properties [Dongarra et al., 1992, Hendrickson and Womble, 1994]:

- Communication volume is within a constant factor of the optimal.
- Higher communication parallelism degree.
- The maximal load imbalance is $N^2(P_r + P_c - 2)/(2P)$, which is small compared to the runtime $\Theta(N^3/P)$.
- The proportional load imbalance stays constant while increasing the number of processors.
- Less processor idle time.
- Matrix size N is capable of growing with $\sqrt{P}$ to maintain scalability while increasing the number of processors given the fact that the matrix memory requirement grows with N^2.

On the other hand, it allows us to design a compact efficient runtime system to avoid complex cases such as distributed work stealing and dynamic tracking of the ownership of blocks. We believe the property of the bounded load imbalance is able to provide us with a nearly balanced workload on every process. Both ScaLAPACK and our experiments have shown that using the 2D cyclic distribution can achieve good load balance and high performance.

6.3.3 Task and task mode

It is sufficient to create a single instance for each task on shared-memory machines. The single instance contains all the necessary information for the runtime system to analyze data dependences and execute the task. Figure 6.4 shows the data structure to store tasks in our runtime system. A task contains the following information:

- Task-related information such as task id, function type, and priority.
- Input: which blocks are the inputs and a ready status for each input. A block is denoted by a 3-tuple (matrix, row index, column index).

```
struct Task {
    int      task_id;
    char     type, priority;
    /* input info */
    char     num_inputs;
    int      inputs[ ];
    bool     inputs_ready[ ];
    /* output info */
    int      output[3];
    int      num_minor_ouputs;
    int      minor_outputs[ ];
    /* distri-system extension */
    char     mode;
    bool     is_generated;
    char     which_input;
};
```

Figure 6.4: The `task` data structure.

- Output: which block is the output. If a task has more than one output, we distinguish them as *minor* outputs.

By contrast, a task on a distributed-memory machine will correspond to a number of task instances since each task will be created and checked by all the processes. We assume each task has $O(1)$ number of inputs and outputs and propose a novel approach to generating tasks. The objective is to make all the processes reach the same conclusion without any cooperative communication.

Suppose a task has c_1 inputs and c_2 outputs, then we create $c_1 + c_2$ task instances and allocate them to different processes. Each task instance plays a role of "representative" for the task's corresponding input or output. The location of the task instance (representing either an input or output) is decided by the location of the input or output. The major output of a task corresponds to an *owner* task instance and it is the process who stores the owner instance that execute the task. Each input of the task corresponds to an *input shadow* task instance (either local or remote). The runtime system utilizes the input shadows to notify the corresponding owner task that a certain input becomes available. If a task has

more than one output, then for each minor output, we either create a *local minor-output-shadow* task instance or a pair of *source/sink minor-output-shadow* task instances. Table 6.1 summarizes the definitions of the six task modes. Note that the location of a task's input or output is well defined by the 2D cyclic distribution function.

6.3.4 The distributed algorithm

In the distributed algorithm, we partition the task list of the centralized algorithm into multiple task lists across different processes. Each process maintains a private task list. To reduce the time to traverse the task list, we further divide a process's private task list into a number of *block access lists* so that each block $A[I, J]$ is associated with a separate task list (Fig. 6.5). This way, we can perform the APPEND and FIRE operations quickly on shorter lists by using the block index $[I, J]$. Section 6.4.3 discusses the memory requirement to store block access lists.

The distributed algorithm predefines an arbitrator for each block to decide the data dependence only involving the block. At any time, only the block's arbitrator makes the unique decision. We let the process that owns the block be the arbitrator and determine data dependences for its owned set of blocks. We extend the previous centralized algorithm to the distributed algorithm using the following rules:

- Both blocks and tasks are allocated to specific processes by 2D cyclic distribution.
- Every process has a task-generation thread and generates tasks independently.

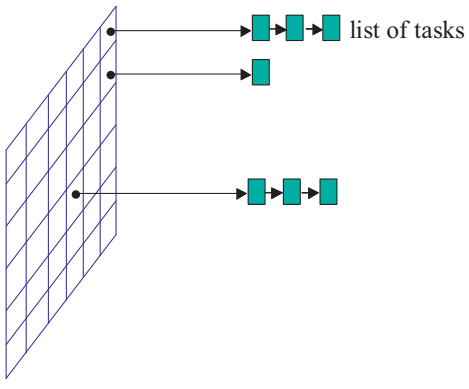


Figure 6.5: Block access list to store tasks in the runtime system.

Table 6.1: A variety of task modes.

Task mode	Definition
Owner	An owner task instance is stored by the process which owns the task's major output. The owner task instance keeps the complete information of the task.
Local input shadow	A local input shadow task instance is stored by the process which owns the specific input. The input block and the task's major output block must belong to the same process. The local input shadow task instance keeps partial information regarding which specific input to read and a pointer to the owner task instance.
Remote input shadow	A remote input shadow task instance is stored by the process which owns the specific input. The input and the task's major output must belong to different processes. The remote input shadow instance keeps partial information about which input to read and what is the output.
Local minor output shadow	A local minor-output shadow task instance is stored by the process which owns the minor output. The minor-output and the task's major output must belong to the same process. The task's owner task instance keeps a pointer pointing to the local minor-output shadow.
Source minor output shadow	If the minor-output of a task belongs to a process different from the task's major output, a source minor-output shadow task instance is generated and stored by the owner task's process. The source minor-output shadow instance keeps partial information regarding what is the minor-output block. The task's owner instance keeps a pointer to the source minor-output shadow.
Sink minor output shadow	A sink minor-output shadow task instance is stored by the process to which the minor output is assigned. The availability of the sink minor-output shadow is notified by the availability of the source minor-output shadow.

- Every process only stores and keeps track of matrix blocks assigned to itself.
- Every process stores "relevant" tasks only. That is, suppose a task t takes block $A[I, J]$ as an input or output, $A[I, J] \in$ process P implies that an instance of task t will be created and stored by P. A specific task mode will be assigned to the task instance based on Table 6.1.

We use a simple example to show how the distributed algorithm works (Fig. 6.6). Suppose a matrix of size 3 blocks by 3 blocks is distributed to a 2×2 process grid by 2D cyclic distribution, then each process is allocated with a set of blocks (i.e., shaded blocks). Let the processes P1, P2, P3, P4 execute a sequential program and generate a set of tasks: t_1, t_2, and t_3. We assume task t_1 reads and writes block 1, t_2 reads block 1 and writes block 4, and task t_3 reads block 1 and writes block 7. Figure 6.6 illustrates which task instances of t_1, t_2, t_3 are generated and where and how they are stored. Since all the blocks accessed are irrelevant to P2 and P4, P2 and P4 have empty task lists. Based on the status of the task lists on P1 and P3, it is easy to find that t_2 and t_3 can be started simultaneously after task t_1 is finished.

Theorem 6.1. *The distributed algorithm guarantees that a task will eventually get all of its inputs and become ready.*

Proof. Suppose a task T $\in P_0$ has k inputs which are allocated on k different processes $\{P_1, \ldots, P_k\}$ (Fig. 6.7). Let task T's ith input be generated by P_i at time t_i and be received by P_0 at time t'_i. Also suppose task T is generated by P_0 at time t_0. There could be an arbitrary order in the set $\{t_0, t'_1, t'_2, \ldots, t'_k\}$. To prove task T can eventually get all of its inputs and become ready, we only need to show the distributed algorithm handles the following three cases correctly:

Case 1: $t_0 < \{t'_1, t'_2, \ldots, t'_k\}$. At time t_0, P_0 creates task T that is stored in P_0's task window with all of T's inputs not ready. Later on, P_i sends T's i-th input to P_0 at time t'_i for which P_0 receives it and marks the i-th input of T as "ready". After all the k inputs arrive, task T becomes a ready task.

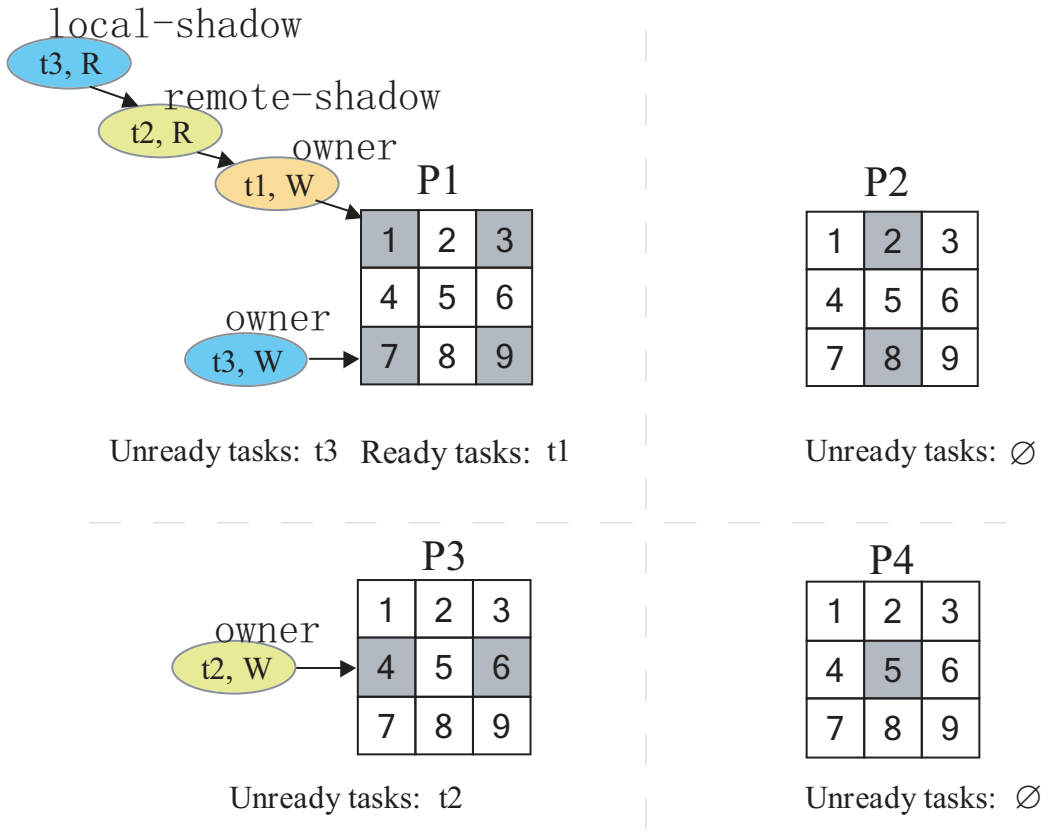


Figure 6.6: Snapshot of the distributed algorithm after four processes have generated tasks t_1, t_2, and t_3 in sequence. Assume t_1 reads/writes block 1, t_2 reads block 1 and writes block 4, and t_3 reads block 1 and writes block 7.

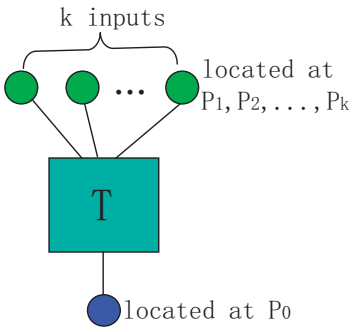


Figure 6.7: A task T with k inputs and 1 output.

Case 2: $\{t'_1, t'_2, \ldots, t'_k\} < t_0$. At time $t_m = min_{1 \le i \le k} t'_i$, P_0 receives an input of task T. Since the owner task instance of T has not been generated by P_0, P_0 creates a partial task instance for T and marks the received input as "ready". The arrival of the other inputs makes P_0 update the status of the other inputs in the partial task. Finally at time t_0, P_0 generates the owner instance of T. It searches for the partial instance, fills in T's complete information, and moves task T to the ready queue.

Case 3: $\{t'_1, \ldots, t'_s\} < t_0 < \{t'_{s+1}, \ldots, t'_k\}$. It is actually equivalent to the mixed case of 1 and 2. That is, $\{t'_1, \ldots, t'_s\} < t_0$ is handled similarly to Case 2, and $t_0 < \{t'_{s+1}, \ldots, t'_k\}$ is handled similarly to Case 1. □

Theorem 6.2. *The distributed algorithm is deadlock free for any task window of size* $W \ge 1$.

Proof. Each process P_i has a task window Q_i. Suppose a deadlock occurs between m processes $\{P_{i_1}, P_{i_2}, \ldots, P_{i_m}\}$ that form a waiting cycle such that P_{i_k} waits for $P_{i_{k+1}}$. If the first task in a task window is a non-owner task, it will be executed and removed immediately. So when the deadlock happens, the first task $\in Q_i$ must be an owner task. Let t_{i_d} be the first task in Q_{i_d}, where $d \in [1, m]$. Suppose $t_{i_1} \in Q_{i_1}$ is unable to execute because t_{i_1} is waiting for one of its parent task $\wp(t_{i_1}) \in Q_{i_2}$ to finish. Task $\wp(t_{i_1})$ must be either t_{i_2} itself or behind t_{i_2} in Q_{i_2}. Thus, $t_{i_2} \le \wp(t_{i_1}) < t_{i_1}$ is the sequential order of the tasks. By following the deadlock cycle of $\{P_{i_1}, P_{i_2}, \ldots, P_{i_m}, P_{i_1}\}$, we can show that $t_{i_1} < t_{i_m} < t_{i_{m-1}} < \ldots < t_{i_2} < t_{i_1}$. It contradicts the fact that each task window Q_i stores the tasks in the program's sequential order. □

6.4 Runtime system design

On a distributed-memory system, every compute node runs an instance of the runtime system. In particular, a single shared-memory machine needs only one runtime system instance. The runtime system has two task pools: a task window and a ready task pool. The task window stores all the generated but not finished tasks. The implementation of the task window actually uses the *block access lists* indexed by block locations $[I, J]$. The maximal number of tasks to be generated is constrained by the task window size. As introduced in Sect. 6.3.4, a process's runtime system only keeps the blocks and tasks that are assigned to the process based on the 2D cyclic distribution. The ready task pool is much simpler than the task window. It only stores a pointer pointing to the corresponding ready task in the task window. After finishing a task, the runtime system uses the pointer to remove the ready task from the task window. Each task has a priority. Hints regarding critical paths (e.g., panel tasks in factorizations) are provided by the task-based library writers and tasks on the critical path are assigned a high priority.

6.4.1 Thread types

The runtime system has three types of threads: task-generation thread, computing thread, and communication thread. If a node has n processor cores, we launch one task-generation thread, one communication thread, and $n-1$ computing threads. We let the $n-1$ computing threads occupy $n-1$ cores, and the communication thread and the task-generation thread occupy the remaining one core.

As shown in Fig. 6.8, the task-generation thread simply executes a sequential program and generates tasks to fill in the task window by invoking the APPEND operation. The task-generation thread uses a counting semaphore to start or stop depending on whether the task window is full or not. When a computing thread becomes idle, it picks up a ready task from the ready task pool and computes it. After finishing the task, the computing thread performs the FIRE operation to solve data dependences and find the children of the finished task.

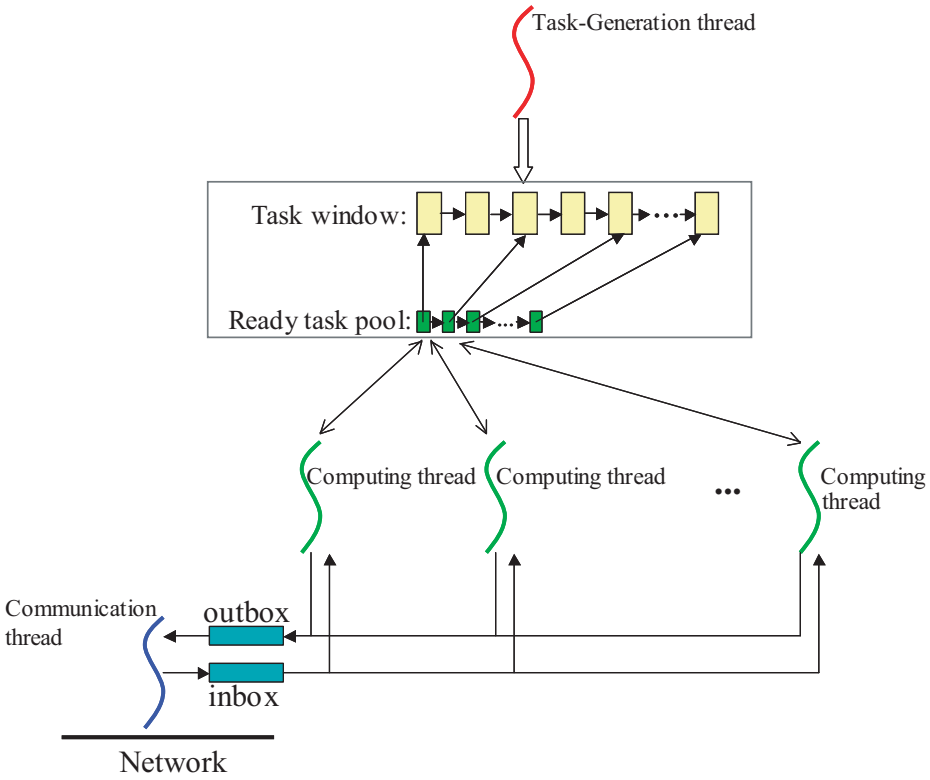


Figure 6.8: Architecture of the runtime system.

The communication thread is responsible for sending and receiving messages by posting `MPI_ISend` and `MPI_IRecv` operations. The interaction between the computing threads and the communication thread is through the message `inbox` and `outbox`. If a computing thread wants to send a block to some computing threads running on different nodes, it puts a message in the `outbox` and the communication thread will send it out. Whenever receiving a message, the communication thread places the message in the `inbox` which will be read by one of the computing threads.

The communication thread and the task-generation thread take the last core. The reason why we do not launch multiple communication threads is because the thread support level of MPI_THREAD_MULTIPLE at the moment is not portable on all systems and mixing computing thread and communication thread on the same core interferes with the maximized cache hit rate of the computing thread. When n is big enough (e.g., 16 or more cores), it is reasonable to dedicate one of the many cores to process communications.

6.4.2 Memory allocation and deallocation

We use an indirect data structure to store matrices. Given a matrix of size N and block size of NB, the indirect data structure consists of $(\frac{N}{NB})^2$ pointers pointing to a number $(\frac{N}{NB})^2$ of NB $\times$ NB blocks for the matrix. Figure 6.9 shows how to use an indirect data structure to store a matrix with 2×2 blocks. There are two matrix types in our runtime system: user-defined input/output matrix and intermediate-result matrix. The intermediate-result matrices are allocated and deallocated on demand by the runtime system. The first task that writes to the intermediate block will allocate memory for the block. Here we assume there is sufficient memory to allocate.

It is more difficult to deallocate blocks because the runtime system itself cannot decide whether a block will be needed or not in the future. Similar to ANSI C programs calling `free()` to release memory, we provide programmers with a special routine `Release_Block()` to free a block. `Release_Block()` actually doesn't release any memory, but sets up a marker in the task window. While generating tasks, the task-generation thread keeps track of the expected number of visits for each block. Meanwhile the computing thread records the actual number of visits for each block. The runtime system will free a block if and only if the following three conditions are satisfied: i) The block is currently stored by the process. ii) The actual number of visits is equal to the expected number of visits to the block. iii) `Release_Block` has been called to free the block. In our runtime system, each data block maintains three data members for the memory deallocation: `num_expected_visits`, `num_actual_visits`, and `is_released`. This memory deallocation method bridges the gap

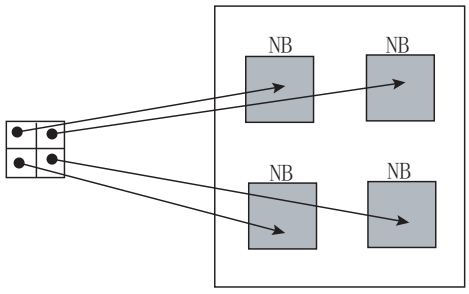


Figure 6.9: Indirect data structure for matrices.

between the on-the-surface deterministic programs and the internal nondeterministic execution by the dynamic runtime system based on data availability.

6.4.3 Space overhead

This section analyzes the memory requirement for the runtime system to keep track of data dependences. For every input and output of a task, there is a task instance generated and added to the task list. The owner task instance stores the complete information of the task. Suppose the owner task has k arguments, then it uses $3k \times 4$ bytes since each argument is represented by 3 integers. A non-owner task instance stores information of task id, block location, and a flag of input or output (i.e., 17 bytes). Therefore, every task corresponds to $12k+17(k-1)$ bytes. Suppose the task window size is equal to W, then it takes $W(29k-17)$ bytes to store the dependence related information.

Although a smaller task window can save more memory space, a larger window size could explore more tasks in a longer distance and identify more parallelism. In our implementation, we choose the window size to be equal to the number of blocks assigned locally to each process. Note a matrix is distributed across processes in the 2D cyclic distribution. Suppose a compute node has a memory of capacity M bytes (e.g., 4 GBytes). Let NB be the block size, then the local matrix has a maximum dimension of $\sqrt{M/8}$/NB blocks and the task window size W is of $M/8/\text{NB}^2$. The ratio of the space for storing the W tasks over M-byte is thus $(29k-17)/8/\text{NB}^2$. To avoid too fine-grained tasks, we expect NB to be at least 32. Suppose a sufficiently complex task has a number $k = 16$ of arguments, the overhead is just about 5.5% for NB=32. The overhead becomes much smaller when NB is bigger and k is smaller. For instance, if k is still 16 but NB=100, the space overhead is equal to 0.56%.

6.5 Analytical analysis

We use the tiled algorithms presented in [Buttari et al., 2009] to implement the Cholesky, LU, and QR factorizations. The Cholesky factorization algorithm has a very high degree of

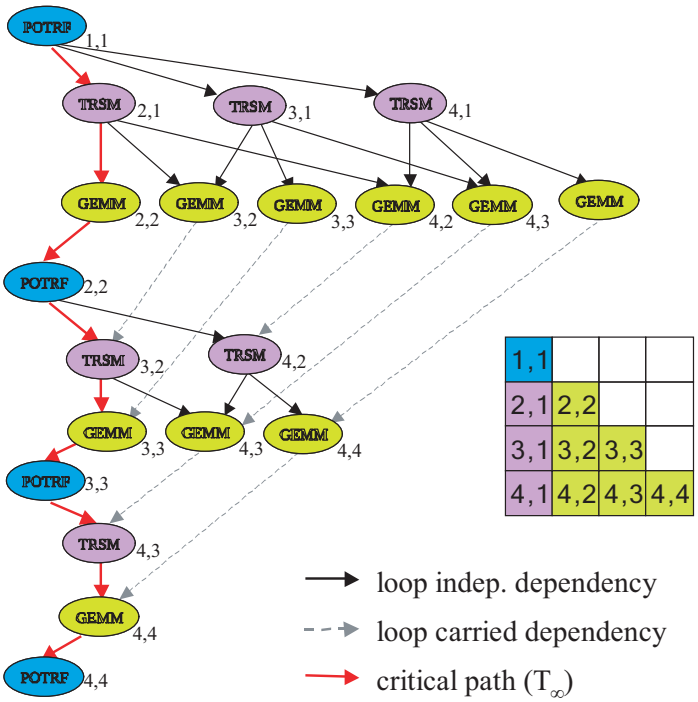


Figure 6.10: DAG for the tiled Cholesky factorization.

parallelism where each finished task in the panel fires a number nb of tasks in the trailing submatrix (nb denotes the matrix dimension in blocks). A finished task `TRSM[i,j]` fires the `GEMM` tasks located on the i-th row or the j-th column in the trailing submatrix. Figure 6.10 shows an example of the DAG for the tiled Cholesky factorization algorithm given a symmetric matrix of size 4 blocks $\times$ 4 blocks.

The tiled LU and QR factorizations use updating-based algorithms whose data dependence graphs are much denser than that of the Cholesky factorization as shown in Figs. 6.11 and 6.12. For details of these algorithms, please refer to [Buttari et al., 2009]. This section skips the simpler Cholesky factorization and analyzes the performance of the updating LU and QR factorization algorithms.

In both LU and QR factorizations, the trailing matrix update occupies the most of the computation. Figure 6.13 shows how to update a trailing submatrix of size 4 blocks $\times$ 4 blocks on a 2 $\times$ 2 process grid. Each task in the i-th row is dependent on the task in the (i-1)-th row and all the tasks on the same row are totally independent.

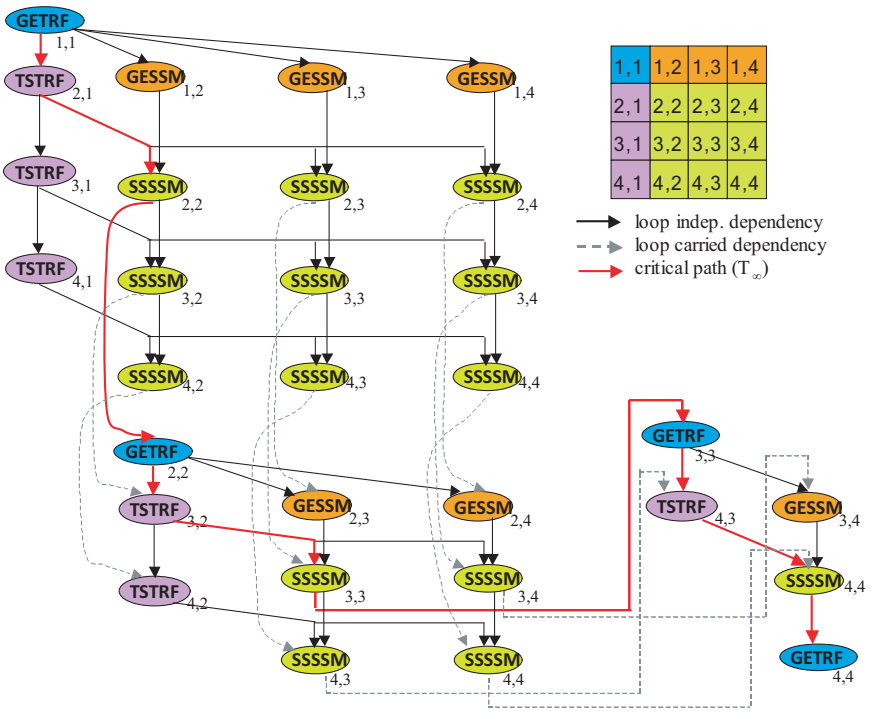


Figure 6.11: DAG for the tiled LU factorization.

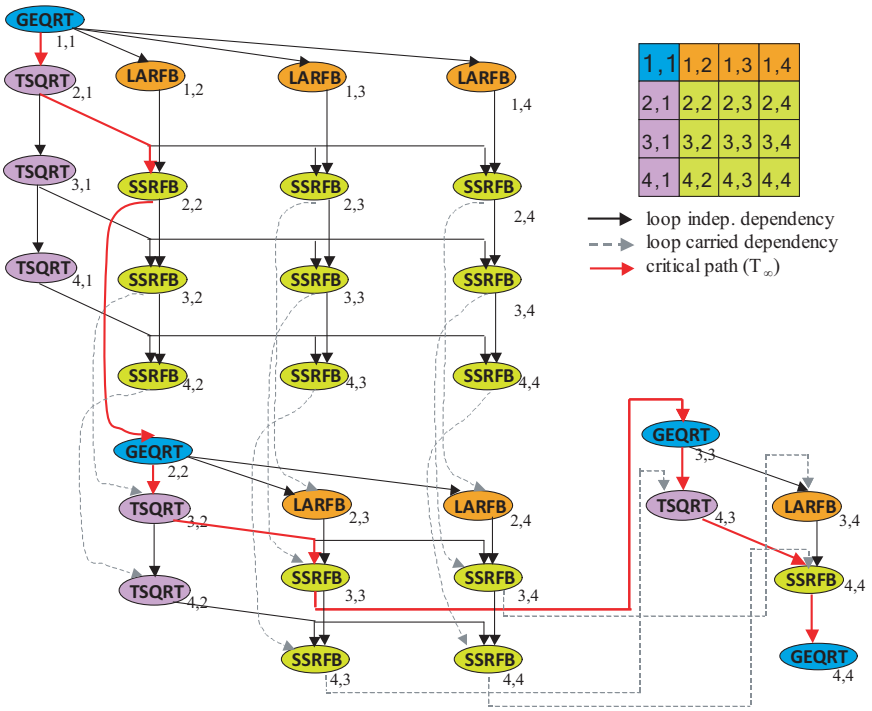


Figure 6.12: DAG for the tiled QR factorization.

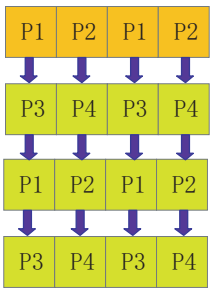


Figure 6.13: Trailing submatrix update in the tiled LU and QR algorithms.

6.5.1 Expected execution time

Theorem 6.3. *Suppose each task takes the same amount time to compute and tasks on the ith row are dependent on tasks on the $i-1$th row. The tasks located on the same row have no data dependences. If $nb \gg P$, the expected execution time is*

$$T = \frac{nb^2(t_{comp} + \frac{t_{comm}}{k})}{P_r P_c} + 2k(P_r - 1)t_{comp} \ , \ where$$

nb is the matrix dimension in blocks, t_{comp} is the computation time of a task, t_{comm} is the communication time to transfer a tile, k denotes a virtual tile of $k \times 1$ tiles, and $P = P_r \times P_c$ is the process grid.

Proof. The tasks in the trailing matrix update are essentially executed along a pipeline. Each process occupies a set of stages in the pipeline (Fig. 6.14), where a process takes charge of a subset of tasks determined by the 2D cyclic distribution.

Consider an arbitrary process P_i. At time $t = 0$, P_i has $nb/P_c \times k$ tasks. The next time P_i appears in the pipeline is at $t + P_r \times (t_{comp} \times k + t_{comm})$ when P_i will get another $nb/P_c \times k$ tasks. Since $nb \gg P$, P_i will get new tasks continuously and never become idle. The expected execution time $T = T_{computation} + T_{communication} + T_{pipestart} + T_{pipefinish}$. $T_{pipestart}$ and $T_{pipefinish}$ denote the elapsed time for all the processes to get the first task to work on. Since process P_i has $\frac{nb^2}{P_r P_c}$ tasks,

$$T_{computation} = (\frac{nb^2}{P_r P_c}) \times t_{comp}, \ T_{communication} = (\frac{nb^2}{P_r P_c \times k}) \times t_{comm},$$

$$\text{and } T_{pipestart} = T_{pipefinish} = k(P_r - 1)t_{comp}.$$

Thus,

$$T = \frac{nb^2(t_{comp} + \frac{t_{comm}}{k})}{P_r P_c} + 2k(P_r - 1)t_{comp}.$$

□

Figure 6.14 shows an example of the pipeline execution on a 4×4 process grid. In the figure, the top-left corner of a large matrix is distributed to the 4×4 process grid by 2D cyclic distribution. For the 4 rows by 4 columns process grid, we only display the first process of each row. That is, $\{P_0, \ldots\}$, $\{P_4, \ldots\}$, $\{P_8, \ldots\}$, and $\{P_{12}, \ldots\}$. Each rectangle in the figure represents a virtual tile that consists of 4×1 tiles. In the example, P0 owns a set of tasks from the first block row at the beginning ($t = 1$). At time $t = 2$, P4 gets its first task. At time $t = 3$, P8 gets its first task, and so on. Similarly at time $t = 5, 6, 7, \ldots$, P0 gets new tasks continuously after P4, P8, P12 finish their tasks in sequence. The same pipeline execution pattern also happens to the other processes.

Corollary. *Let T_1 be the total computation time (i.e., $nb^2 \times t_{comp}$). If $nb \gg P$, then the expected execution time*

$$T = \frac{T_1}{P}(1 + \frac{t_{comm}}{kt_{comp}} + \frac{2k(P_r - 1)P}{nb^2}) \cong \frac{T_1}{P}(1 + \frac{t_{comm}}{kt_{comp}}).$$

Note that $\frac{T_1}{P}$ is a lower bound of the optimal execution time.

6.5.2 Communication and computation ratio

This section analyzes what would be the possible value of $\frac{t_{comm}}{kt_{comp}}$. In the tiled linear algebra algorithms, a task often computes a Level-3 BLAS operation. The Level-3 BLAS operations have a time complexity of $O(\mathrm{NB}^3)$ but access data of space $O(\mathrm{NB}^2)$.

Our analysis assumes the most commonly used BLAS operation of `DGEMM` that has a time complexity of $2\mathrm{NB}^3$. Other Level-3 operations may have different complexities such

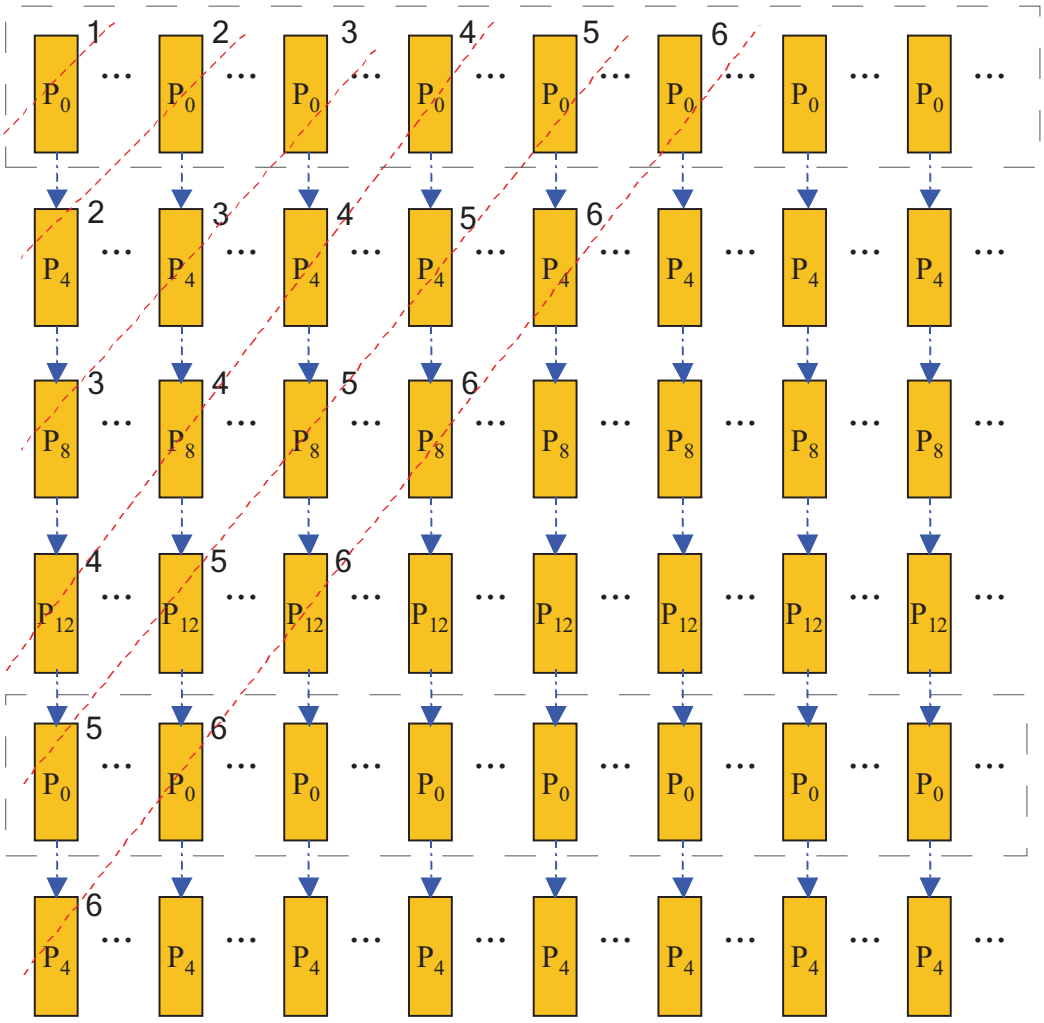


Figure 6.14: Asynchronous pipeline execution by a 4×4 process grid.

as $\frac{1}{3}$NB^3, $\frac{2}{3}$NB^3, and so on. The formula of $\frac{t_{comm}}{kt_{comp}}$ for `DGEMM` is expressed as follows:

$$\frac{t_{comm}}{kt_{comp}} = \frac{8 \times \text{NB}^2/\text{bw}}{2\text{NB}^3 \times k/\text{flops}}, \text{ where}$$

bw denotes the injection network bandwidth in GB/s, and **flops** denotes the maximum `DGEMM` performance in GFLOPS on a single core.

Here we analyze $\frac{t_{comm}}{kt_{comp}}$ for two real machines. The first machine is a cluster machine connected by a Myrinet network. Each core on the cluster has a maximum `DGEMM` performance of 10 GFLOPS and an injection network bandwidth of 1.2 GB/s. The second machine is a Cray XT4 machine, where flops=8 GFLOPS and bw=4 GB/s. As shown in Fig. 6.15, to keep the ratio $\frac{t_{comm}}{kt_{comp}} < 10\%$ on the cluster machine, we should set NB $\geq$ 320 for k=1, NB $\geq$ 160 for k=2, and NB $\geq$ 80 for k=4, accordingly. On the Cray XT4 machine (Fig. 6.16), setting NB=80 and k=1 is sufficient to attain a small ratio of 10%. Thus, users need to adjust NB and k to achieve good program performance on different systems.

6.5.3 Parallelism degree sufficiency

As described in Theorem 6.3, each process in the pipeline receives new tasks continuously if $nb \gg P$. When P is small (e.g., 100's), the condition of $nb \gg P$ can be easily satisfied. In practice, when users double the number of P, the amount of physical memory doubles

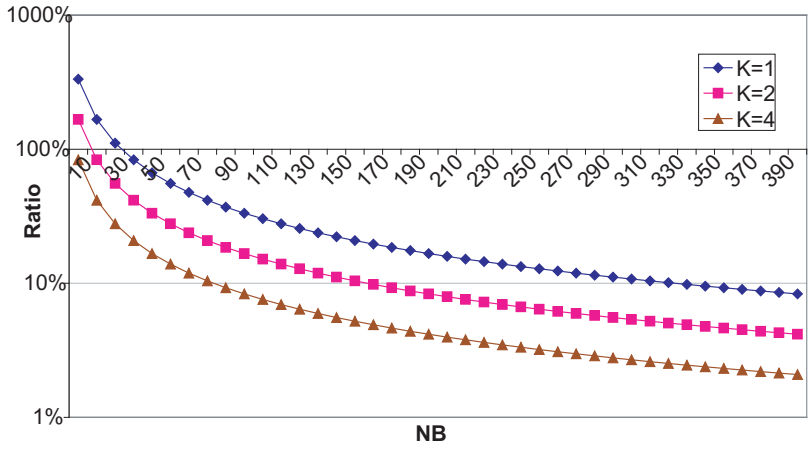


Figure 6.15: $\frac{t_{comm}}{kt_{comp}}$ ratio on a Myrinet cluster.

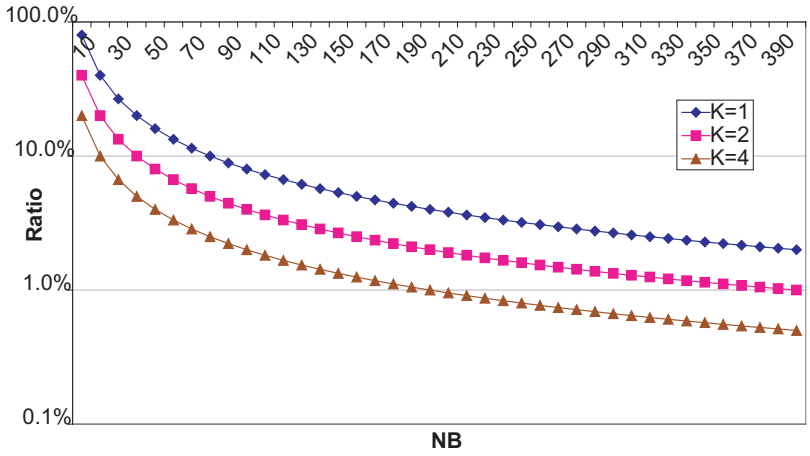


Figure 6.16: $\frac{t_{comm}}{kt_{comp}}$ ratio on Cray XT4.

too. But the matrix dimension N can only be increased by $\sqrt{2}$ and so is nb. The issue we are confronting now is that nb/P will become smaller and smaller when users increase the number of nodes.

This section discusses the condition for which the tiled updating algorithms can achieve good scalability if we double the number of processor cores constantly. Suppose a matrix of dimension nb blocks is distributed across a $P = P_r \times P_c$ process grid, and each process has C threads running on C processor cores. The updating algorithm executes a number nb of iterations. In the ith iteration, process p appears in a number $(nb - i)/P_r$ of rows. Also process p appears $(nb-i)/P_c$ times on each row. The tasks within each row are totally independent and can be executed in parallel, but the tasks between rows must be executed in sequence. We let *TaskGain* represent the total number of tasks within process p. Between every two appearances of process p on two rows, there are a maximal number $\frac{(nb-i)d}{P_c}$ of tasks entering process p assuming a lookahead depth of d. But the C threads in process p also computed (or consumed) $P_r \times C$ tasks since the distance between p's two appearances is P_r tasks and p has C threads. Therefore, in the ith iteration,

$$TaskGain_i = \frac{(nb - i)d}{P_c} - P_r \times C.$$

Because the algorithm has nb iterations and $i = 0 \ldots nb-1$, the overall task gain of process p can be expressed as:

$$TotalTaskGain \cong \frac{nb^2}{6P}(2(1+d)nb - 3 \times P \times C).$$

Note that as long as $(1+d)nb \geq 1.5 \times TotalNumberCores$, every core will keep receiving new tasks constantly and not become idle. We can also conclude that the lookahead technique is able to improve the degree of parallelism and is necessary for the asynchronous algorithms.

6.5.4 Network bottleneck

If there are many cores in a node sharing the same network interface card (NIC), we must make sure the NIC will not become a bottleneck for our algorithms. Suppose a node has n processor cores, and each task takes $\Theta(\text{NB}^3)$ time to compute a block of size NB $\times$ NB. During the time of $\Theta(\text{NB}^3) \times k/\text{flops}$, n cores generate n blocks to communicate. And it takes $8\text{NB}^2 \times n/bw$ time for the NIC to send or receive the n blocks.

In order for the NIC not to become a bottleneck, we require $\Theta(\text{NB}^3) \times k/\text{flops} \geq 8\text{NB}^2 \times n/bw$. That is, $\frac{\Theta(\text{NB}) \times \text{bw} \times k}{8 \times \text{flops} \times n} \geq 1$. For a manycore system connected by a slow network, we need to choose a larger NB to avoid message delays.

6.6 Experimental results

To evaluate the effectiveness of our dynamic scheduling approach and runtime system design, we conducted experiments on both shared-memory and distributed-memory multicore machines.

6.6.1 On shared-memory multicore system

We applied our runtime system to the Cholesky factorization and the QR factorization on two different multicore SMP machines: a 16-core Intel Tigerton machine and a 32-core IBM Power6 machine.

Figures 6.17 and 6.18 show the measurements for Cholesky and QR factorizations on an Intel Tigerton machine with 16 2.4-GHz cores (4 sockets, 4 cores each socket) and 32GB memory. We compiled our dynamic scheduling programs (called `TBLAS`) with Intel Fortran and C/C++ 11.0 compilers at optimization level `-O3`. We compared TBLAS to four libraries: LAPACK, ScaLAPACK, Intel MKL 10.1, and PLASMA. To get a feeling of the performance upper bound, we list the DGEMM performance using 16 cores and 16 times the performance of the serial DGEMM (labeled as 16 × dgemm-seq). For Cholesky factorization (Fig. 6.17), TBLAS is slightly better than Intel MKL but not as good as PLASMA. LAPACK and ScaLAPACK don't provide a good performance on the shared-memory multicore machine. For QR factorization (Fig. 6.18), TBLAS is comparable to PLASMA and both TBLAS and PLASMA are much better than Intel MKL. PLASMA and TBLAS use the same tiled updating algorithms. Since PLASMA uses hand-tuned static scheduling and avoids the runtime system overhead, it is faster than TBLAS.

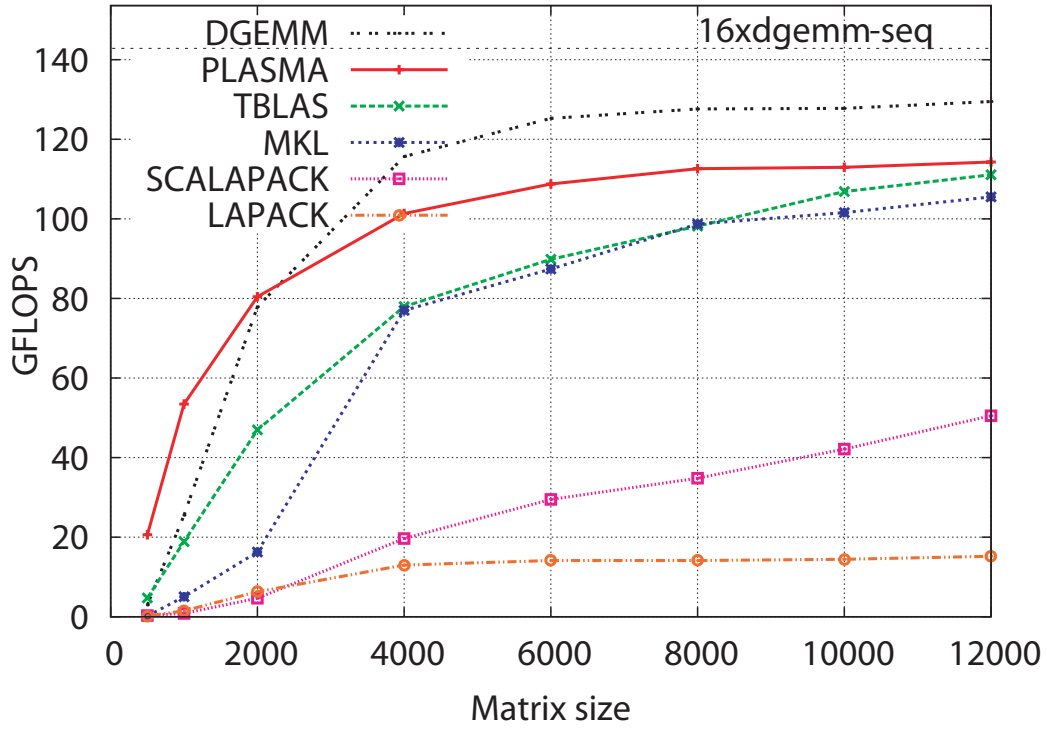


Figure 6.17: Cholesky factorization on a 16-core Intel Tigerton machine.

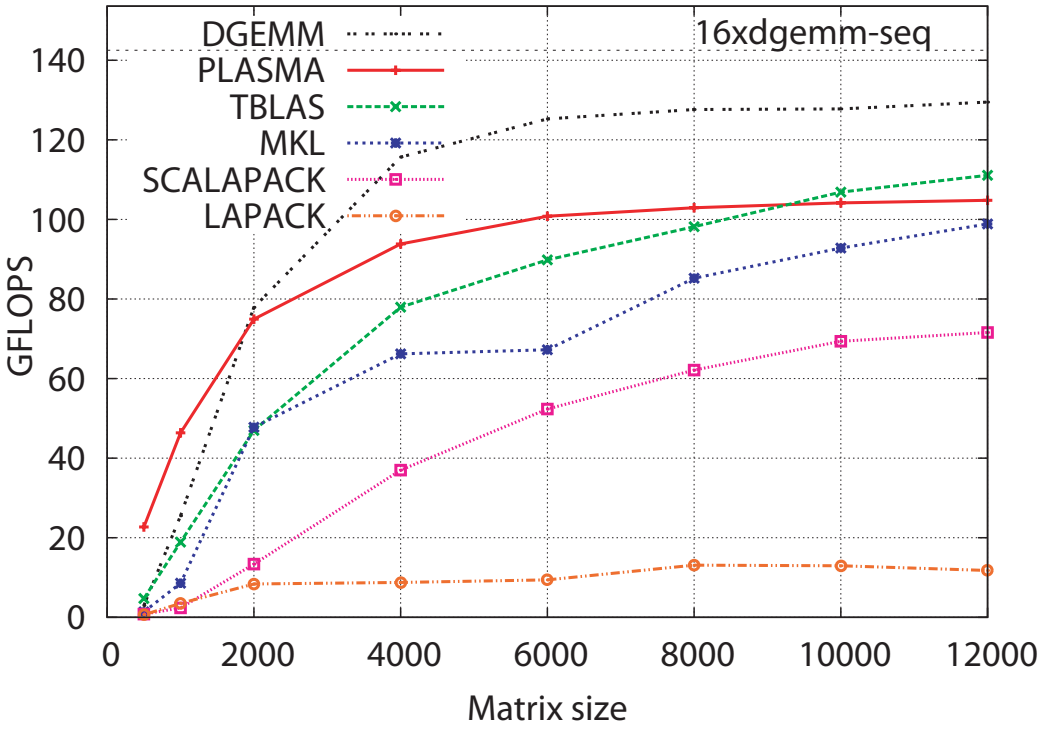


Figure 6.18: QR factorization on a 16-core Intel Tigerton machine.

The second shared-memory multicore system is an IBM Power6 machine with 32 4.7-GHz cores (4 Multi-Chip Modules (MCM), 4 dual-core chips on each MCM). We compiled TBLAS programs with IBM `xlf 11.1` and `xlc 9.0` compilers. Figures 6.19 and 6.20 compare the performance of TBLAS to five libraries: LAPACK, ScaLAPACK, IBM ESSL 4.3, IBM PESSL 3.3, and PLASMA. The performance of DGEMM is also presented to show the performance upper bound. For both Cholesky and QR factorizations (see Figs. 6.19 and 6.20), TBLAS provides a significantly better performance than IBM's ESSL and PESSL libraries.

6.6.2 On distributed-memory multicore system

We measured the performance of our runtime system on the Cray XT4 Jaguar machine from ORNL. The machine consists of nearly 8000 compute nodes each of which has a quad-core 2.3-GHz AMD Opteron processor and 8GB memory. Cray XT4 adopts a 3D torus topology and is connected by a SeaStar2 network. On this machine, the peak performance

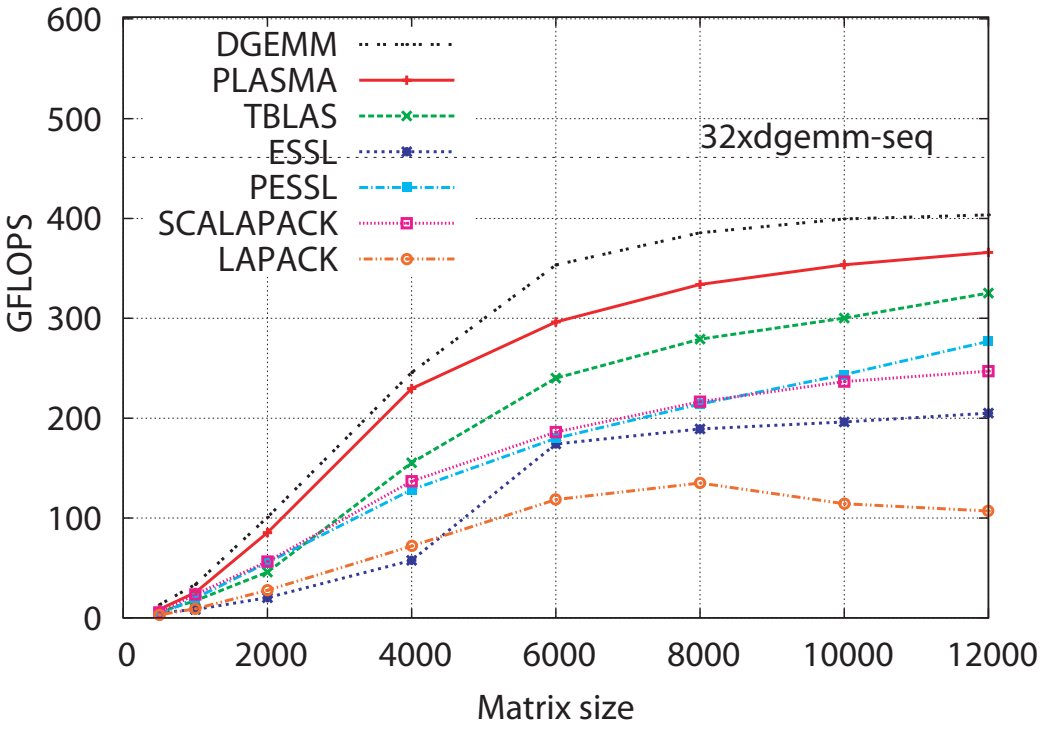


Figure 6.19: Cholesky factorization on a 32-core IBM Power6 machine.

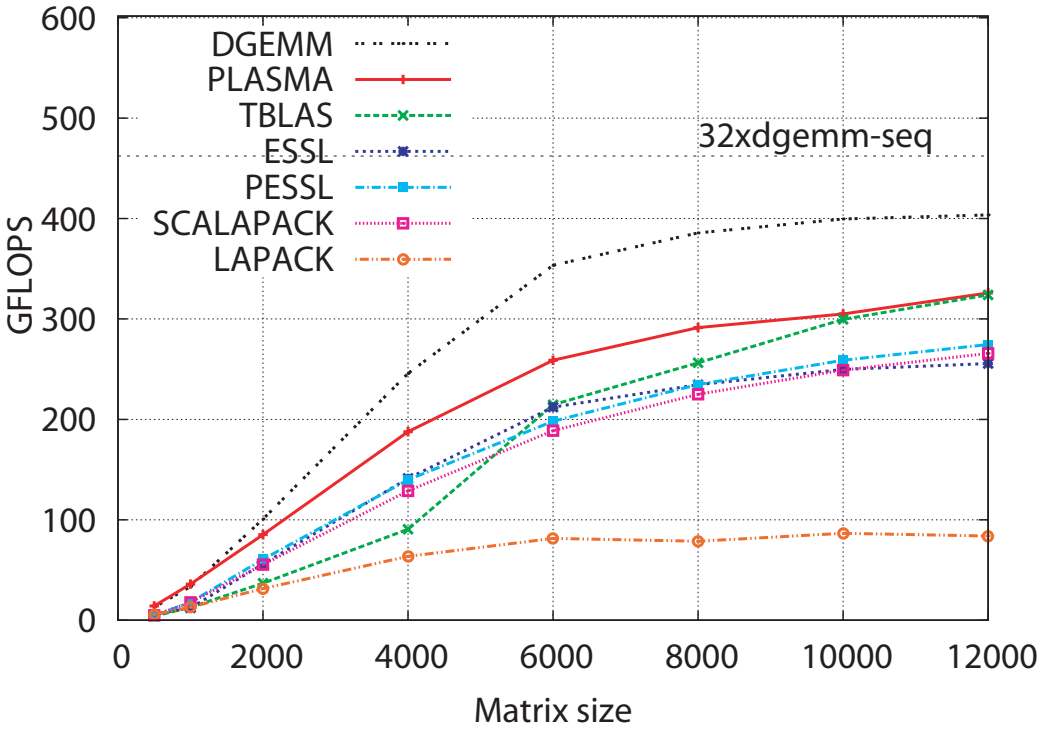


Figure 6.20: QR factorization on a 32-core IBM Power6 machine.

per core is 9.2 GFLOPS and the maximum DGEMM performance per core is 7.6 GFLOPS. We will look at the weak scalability performance of our runtime system. That is, when we double the number of nodes, we also increase the matrix size N accordingly. Since the matrix memory requirement grows with N^2 but the physical memory size grows linearly with the number of nodes, we increase the matrix size by $\sqrt{2}$ when we double the number of nodes. The first matrix size for our single node experiment is 20000. To minimize message delays in our runtime system, we dedicate one core of each node to do nothing but MPI communications. Therefore, we just used 3 out of 4 cores (25% less) on each node to do real computations. However, for the per-core performance, we divide TBLAS's overall performance by 4× NumberNodes, instead of 3× NumberNodes. We believe a node with more than 16 cores can achieve a much better performance (e.g., 1/16=6.25% less).

We compare TBLAS to the ScaLAPACK library provided by Cray `XT-LIBSCI` 10.3.2. We implemented three types of matrix factorizations: Cholesky, LU (with and without pivoting), and QR. For each factorization, TBLAS uses two configurations. One is for the single shared-memory node which uses 4 cores for computation. The other is for the multi-node distributed-memory configuration where each node uses 3 cores for computation and 1 core for communication. Therefore, in the following scalability performance figures such as Fig. 6.22, TBLAS has two different lines for the two configurations, respectively.

Figures 6.21 and 6.22 demonstrate the overall performance and the per-core performance of the Cholesky factorization. The per-core performance is obtained by dividing the overall performance by the number of cores listed on the x-axis. Although TBLAS uses 25% less cores than ScaLAPACK for computation, it is very close to ScaLAPACK (see Fig. 6.21). In Fig. 6.22, the TBLAS `4-comp-core` performance drops from 6.8 GFLOPS to the `3-comp-core` 5.2 GFLOPS performance, which is 23% less than the `4-comp-core` performance. Running on more than one compute node, TBLAS is scalable from 8 cores to 1024 cores.

Figures 6.23 and 6.24 present the experimental results of the LU factorization. We also list the LU factorization without pivoting to compare with the pivoting algorithm. Since LU without pivoting uses an algorithm similar to the Cholesky factorization, its performance

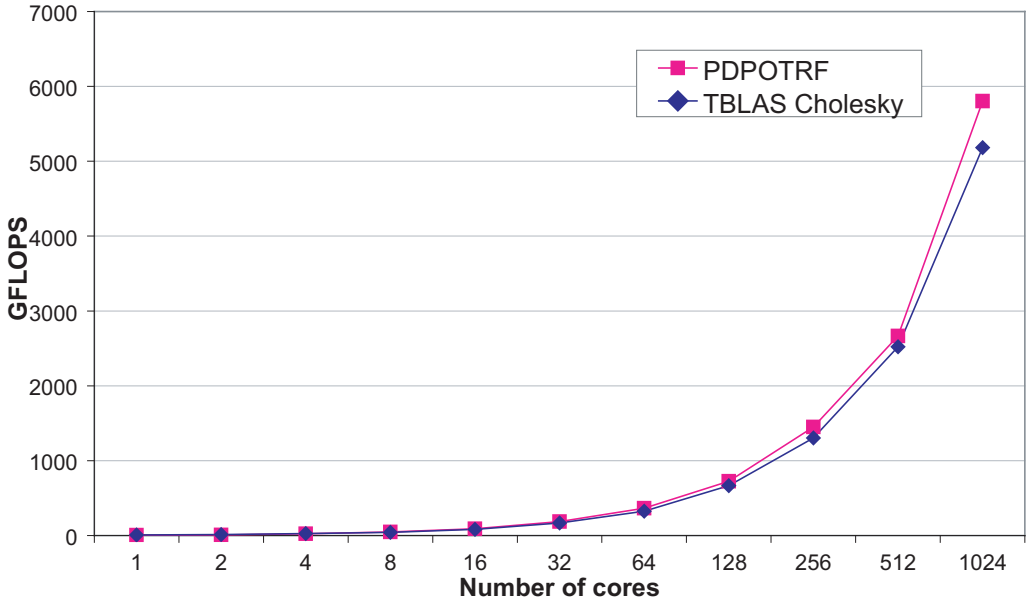


Figure 6.21: Overall performance of Cholesky factorization on Cray XT4.

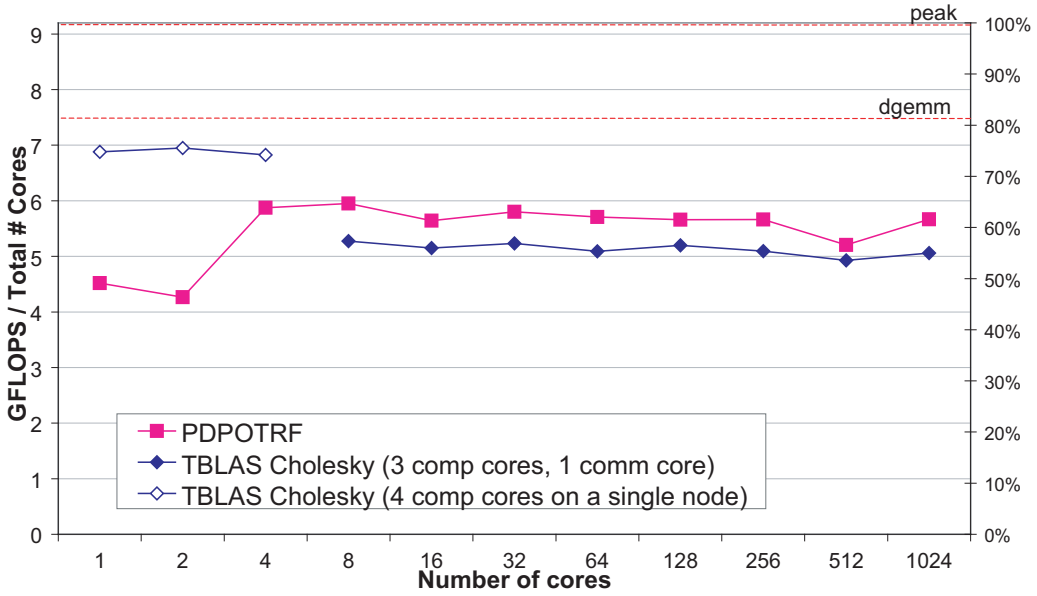


Figure 6.22: Scalability of Cholesky factorization on Cray XT4.

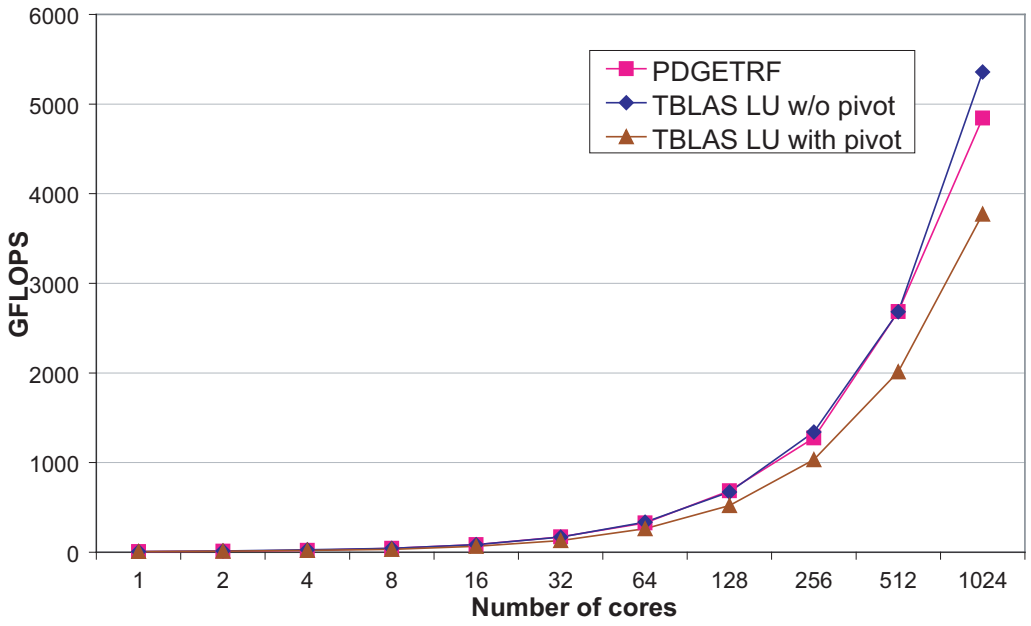


Figure 6.23: Overall performance of LU factorization on Cray XT4.

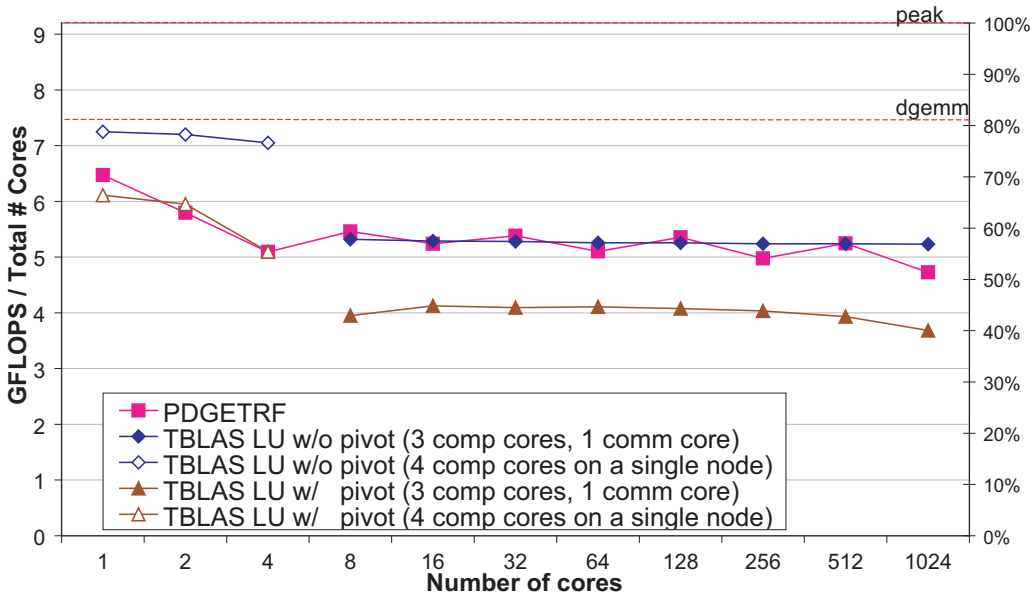


Figure 6.24: Scalability of LU factorization on Cray XT4.

is as good as that of Cholesky factorization. As shown in Fig. 6.24, the LU with pivoting again scales well from 8 to 1024 cores. Moving from `4-comp-core` (i.e., 5.1 GFLOPS) to `3-comp-core` (i.e., 4 GFLOPS), the TBLAS performance is decreased by 21%. The performance of LU with pivoting on a single node drops greatly from two cores to four cores. We are currently looking at how to tune certain parameters in the LU kernels to improve its performance.

Figures 6.25 and 6.26 show the performance of the QR factorization. As seen in Fig. 6.26, the difference between TBLAS `4-comp-core` and `3-comp-core` is equal to 25% (5.7 GFLOPS vs 4.25 GFLOPS), which is just equal to the 25% less compute cores that TBLAS uses than ScaLAPACK.

In summary, our experiment scales from 1 core to 1024 cores on a quad-core distributed-memory machine. At first glance it might appear that TBLAS is inferior to ScaLAPACK. But if a compute node has more than 16 cores, we expect that the TBLAS Cholesky factorization will provide a much better performance than ScaLAPACK since the present 25% unutilized compute cores (1 out of 4 cores) will become less than 6.25% (Fig. 6.22). Similarly, we expect the performance of the TBLAS LU and QR factorizations to be comparable

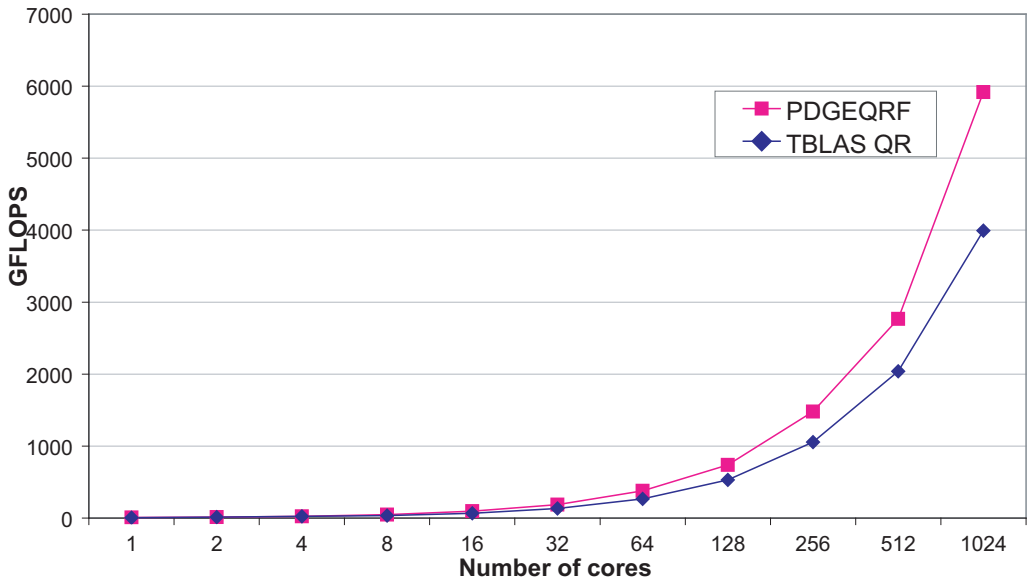


Figure 6.25: Overall performance of QR factorization on Cray XT4.

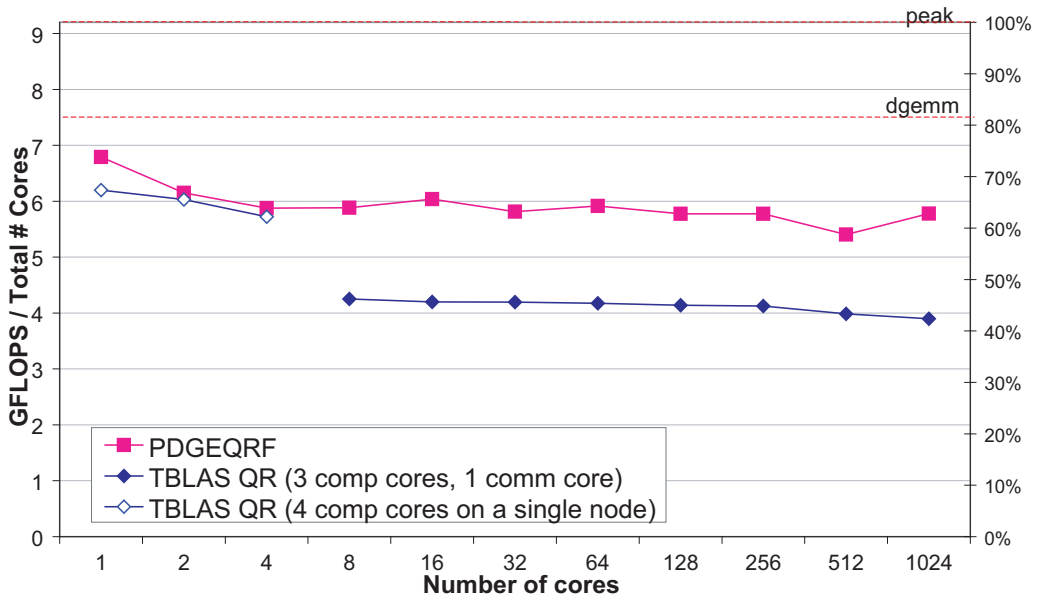


Figure 6.26: Scalability of QR factorization on Cray XT4.

to ScaLAPACK's performance since using more than 16 cores per node will make TBLAS LU and QR factorizations reach the single node performance (Figs. 6.24 and 6.26). Furthermore, the same TBLAS program is able to provide a better performance than ScaLAPACK on shared-memory machines, as shown in Figs. 6.17-6.20.

6.7 Summary

We design a runtime system to schedule tasks dynamically on both shared-memory and distributed-memory multicore systems. Linear algebra programs are written with a task-based library and can be executed by the runtime system automatically. To achieve scalability, the runtime system only stores a portion of the task graph in memory. The runtime system uses distributed block access lists to keep track of data dependences efficiently. To solve data dependences without process cooperation, we design a distributed algorithm for all the processes to make uniform decisions by using six different task modes. We have proved that the tiled algorithms have sufficient degree of parallelism and can offer scalability guarantees. Our experimental results on both shared-memory and distributed-memory machines also demonstrate that the runtime system is scalable.

Chapter 7

Scalable Multicast for Distributed DAG Scheduling

7.1 Introduction

The most common communication used to execute DAGs on a distribute-memory system is multicasting where a completed task must notify its descendants that are blocked awaiting its output. This chapter studies how to perform communication in a scalable way for the distributed dynamic DAG scheduling. MPI libraries provide users with optimized broadcast operations on nearly all high performance machines. Due to the dynamic and irregular parent/children relationship in general DAGs, there could be 2^N possible subsets of processes for broadcasting, where N is the number of processes. For every finished task and its corresponding children, one has to call `MPI_Comm_create()` followed by `MPI_Bcast` to realize the multicast. Even if we ignore the time to create the communication groups, multiple MPI broadcasts involving the same process have to be executed in a sequence because MPI broadcast is a collective operation. Figure 7.1 shows an example where P3 is involved in three communication groups with broadcast roots $P0$, $P1$, $P2$, respectively.

The material from this chapter was published in the 9th International Conference on Computational Science (ICCS 2009) [Song et al., 2009a].

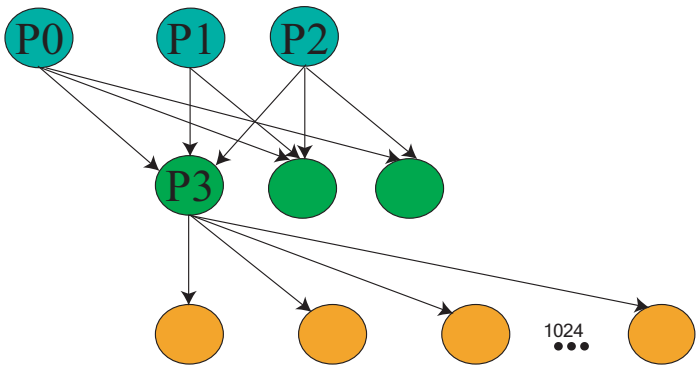


Figure 7.1: Data multicast from parent to children in a DAG.

This chapter presents a scalable multicast scheme to enable dynamic DAG scheduling on large-scale distributed systems with tens of thousands of processors. The multicast scheme assigns a topology ID to each process and builds a compact neighbor table. The scheme is non-blocking, topology-aware, scalable, and deadlock-free, and it supports multiple concurrent multicasts. We compare its performance to the flat-tree multicast and the vendor `MPI_Bcast`. Based on the experimental results, our multicasting scheme is significantly better than the simple flat-tree multicast method and comparable to the vendor optimized collective MPI broadcast.

Different from the automated data dependence solver introduced in Chapter 6, this chapter uses a different method to represent DAGs in a symbolic way. Although the symbolic method has less overhead and is more efficient, it requires a lot of effort for developers to define the correct data dependence relationship for all the algorithms. However, the corresponding multicast method presented in the chapter is generic and can be applied to other distributed DAG scheduling problems easily.

7.2 Computation model

7.2.1 Symbolic task graph

A large number of scientific applications have a loop nest control structure, where both loop bounds and array subscript expressions are affine functions. In such a program, every task

(or statement) is enclosed by a number of iterations. Although the program has a small number of task types, executing the program will unroll the loops and generate a large number of task instances from each type of task. One could expand all the loops and store the task instances into a DAG from the first iteration to the last one. But it is too expensive to store such a DAG and is not scalable. Instead, we represent the semantics of programs with loop nest control structures by polyhedrons such that each task instance corresponds to a unique *coordinate* or *iteration vector*. A task instance is denoted by a tuple `(type, iteration vector)`. By identifying data dependences between tasks, we are able to construct the whole DAG. Similar to the method introduced by [Cosnard and Jeannot, 1999], we define a task graph symbolically as follows: $G = \langle T, E \rangle$, where

$$T = \{\text{task t } : t = (type, \vec{u})\}.$$

The set of edges E are defined by a collection F of symbolic functions:

$$F = \{f_i \text{ for certain task type } i\} \text{ and } f_i : \{\vec{u}\} \rightarrow T.$$

Given a task instance $(t, \vec{u})$, $f_t(\vec{u})$ defines a set of tasks that are dependent on task $(t, \vec{u})$.

Figure 7.2 depicts a DAG example for Cholesky factorization. There are three types of tasks and the data dependence relationship is represented by three symbolic functions. For instance, if a task `p = (POTRF, <k, i, j>)` completes, p's children can be determined by the following function:

$$f_1(p) = \{(type, \langle k', i', j' \rangle) : type = \text{TRSM}, k' = k, i' \in [k+1, n], j' = k\}.$$

Similarly, `TRSM` and `GEMM` each has its own symbolic function to define the children.

7.2.2 Programming model

We design a simple application programming interface (API) to defines DAGs in a symbolic way. After the user implements the API routines, the underlying runtime system can

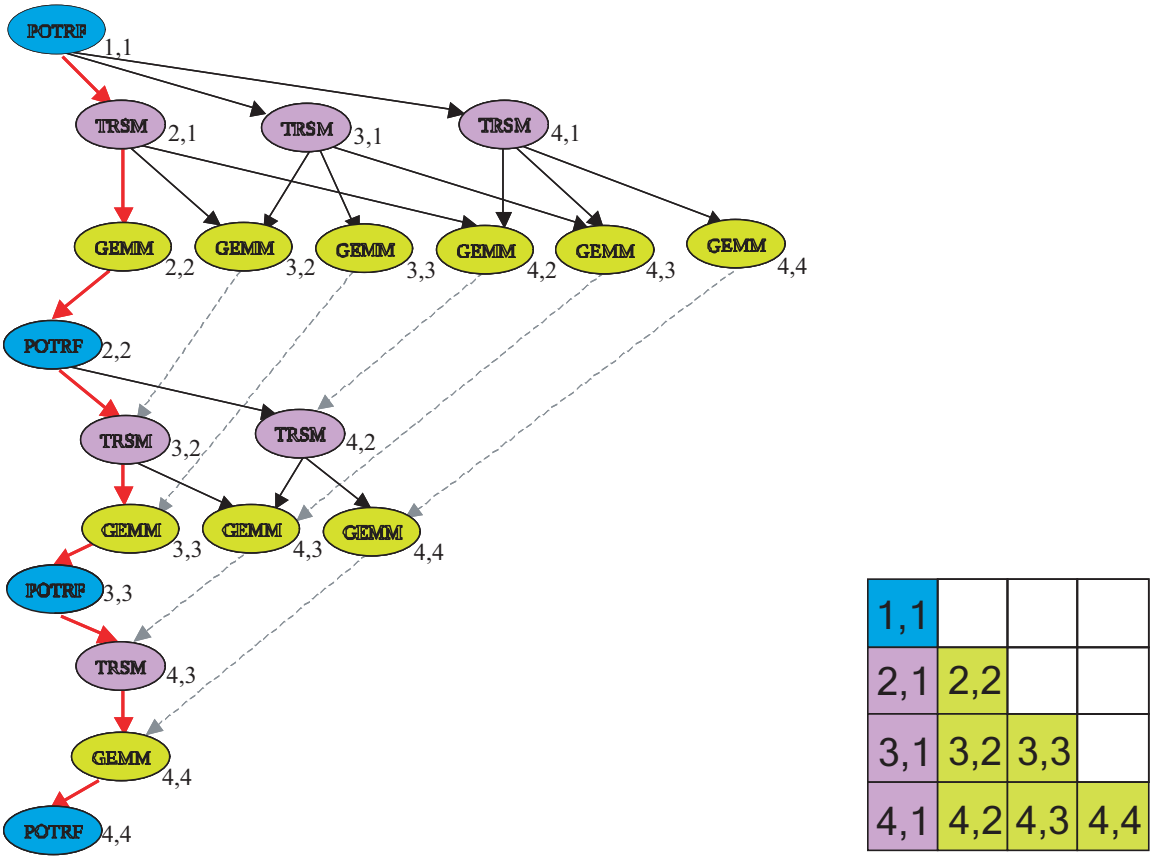


Figure 7.2: Cholesky factorization DAG for a matrix with 4×4 blocks.

automatically parallelize and execute the DAG on shared- or distributed-memory systems. The ANSI C programming interface routines are listed below:

```
int  get_children(const Task t, Task children);
int  get_num_parents(const Task t);
void set_entry_task(const Task t);
void set_exit_task(const Task t);
```

Note that we can obtain the child tasks easily by calling `get_children()` given a finished parent task. As long as each member of the multicast group is notified of the parent task, it is able to deduce the whole group immediately. If the `get_children()` function is not feasible, the group members have to be included explicitly in messages.

We defined the user interface for Cholesky factorization and conducted experiments on a cluster machine. The implementation of the interface has about 40 lines of code. The cluster has 64 dual-CPU (Intel Xeon 3.2GHz) compute nodes and is connected by Myrinet. Figure 7.3 shows the weak scalability performance of our program compared with ScaLAPACK. The program's runtime system is the same as the runtime system described in Sect. 6.4 except for the symbolic DAG representation.

7.3 Multicast scheme overview

When a set of processes are executing a DAG, multiple sources may want to notify different groups of children simultaneously. Our new multicast scheme is able to provide this functionality automatically. The scheme is essentially an application-level routing method. Every process owns a compact routing table. Although each process only has knowledge of a few local neighbors, the whole group of processes is represented by a collection of hierarchical trees. Most importantly, every process is the root of its own multicast tree. The routing algorithm simply follows the tree to multicast data to a set of children.

In Fig. 7.4, the process on compute node `001` wants to multicast data to $\{010, 100, 101\}$. For the system with eight compute nodes, it takes three steps to complete the multicast. There could be at most eight processes running (at the leaf level) on the system, but some

processes will be mapped to serve as "virtual masters" responsible for their corresponding subtrees. Our method to build the routing table guarantees that there exists a path from the source to every destination by filling in "virtual masters" (Lemma 7.1 in Sect. 7.8). The path length is bounded above by $logN$. The multicast scheme always starts from a root to a set of leaves and every process has its own multicast tree with itself as the root.

7.4 Topology ID

To improve communication performance, it is critical to know the communication cost between any pair of nodes on a system. One could build a $N \times N$ table to describe the latency and bandwidth information between every pair of nodes. But for a system with millions of nodes, it is too costly to build and maintain such a big table. Another natural approach is to use hierarchy as an abstraction to achieve scalability. The hierarchy abstraction has been widely used on the Internet, for example for DNS and IP addresses, as well as for message passing operations on computational Grids. It is also very common in our daily life, for example, the composition of a country and the postal ZIP code.

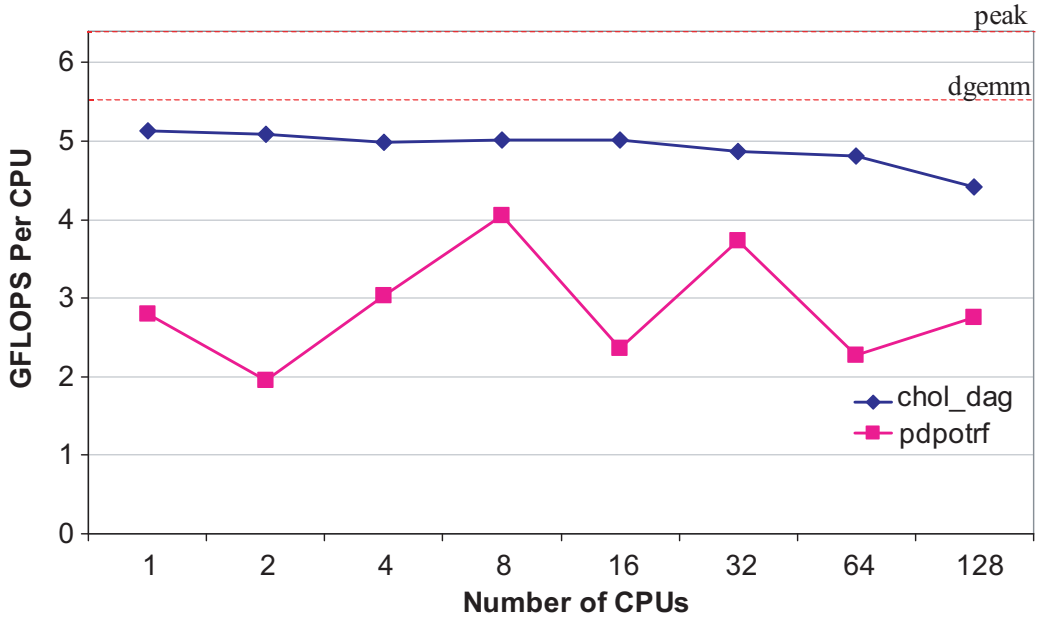


Figure 7.3: Cholesky factorization on a cluster machine.

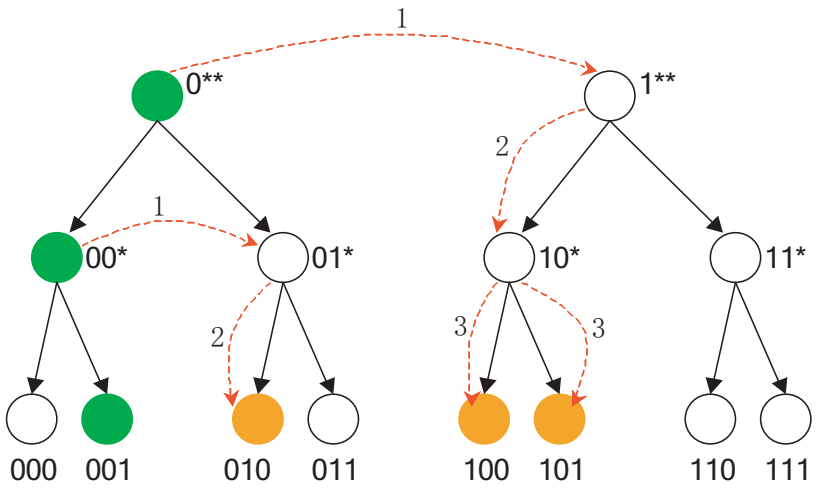


Figure 7.4: Multicast on a system with 8 compute nodes.

Instead of exposing all the other nodes to the source, the hierarchy technique utilizes a hierarchical tree to send data level by level. This way each node only communicates with a small number of nodes so that the system keeps scaling. A high performance computer could build upon racks that consists of nodes. A node has a set of boards. Every board has a number of multicore chips on which there are tens or hundreds of processor cores. Typically communication cost on the lower level (e.g., cores on the chip) is less than that on the upper level (e.g., cores on different nodes). It is also easy to find out how to place every core in a hierarchical tree.

We assign a topology ID to each compute node on the system. Similar to ZIP codes, we assume that the longer the common prefix of the two nodes' topology IDs, the closer they are and the smaller the latency. When a process is running on a node with topology ID x, we say the process has topology ID x. As shown in Fig. 7.5, the SGI Altix 3700 system (128 processors) is comprised of 8 groups, each group has four C-bricks. Every C-brick has four processors. A toplogy ID of format $< 3bits, 2bits, 2bits >$ is sufficient to identify which group and which C-brick a processor belongs to. Given any two topology IDs, it is easy to decide whether they are close or not. This assignment of topology IDs reflects the latency between processors on the SGI machine correctly. For other topologies such as ring and mesh, we need to enforce the hierarchy relationship to the physical network. Since it

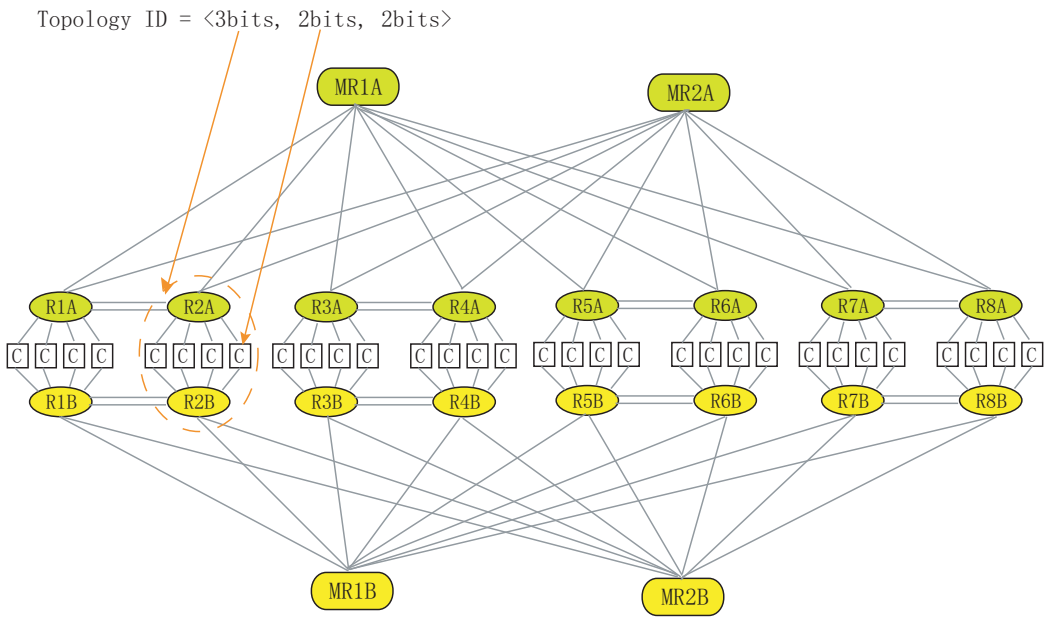


Figure 7.5: An example to assign topology IDs to an SGI Altix 3700 System with 128 processors. C denotes a 4-processor C-brick.

is unusual for a system to change its interconnection network frequently, we assume the topology IDs are static information, and are precomputed and stored in advance under a system directory.

7.5 Extension to Plaxton's neighbor table

In this section, we briefly describe Plaxton's neighbor table and our extension to support multicasting. Plaxton uses an incremental routing approach similar to hypercube routing which resolves the destination node address dimension by dimension [Plaxton et al., 1997]. Supposes a system has $n = 2^m$ nodes, where m is a multiple of b. Plaxton assumes that each node has a label that is independently and uniformly distributed at random between 0 and $n - 1$. Instead of using a random label for each node, we assign a topology ID $\in [0...n - 1]$ to each node. The topology ID reflects the latency relationship (near or far) between two nodes, and is expressed as a sequence of $\frac{m}{b}$ digits with the base 2^b. For instance, if one system has $4096 = 2^{12}$ nodes, base $= 2^3$ leads to a 4-digit octal topology ID.

Every node has its own neighbor table T. Table T consists of r rows and c columns, where r is equal to the number of digits ($=\frac{m}{b}$) and c is equal to the digit's base ($=2^b$). Table entry $T[i,j]$ stores the forwarding node address. In our application-level multicasting, we store an MPI rank as the forwarding address. Let node x have a topology ID of $id^{(x)} = d_0 d_1 \dots d_{r-1}$. If table entry $T[i,j]$ in node x contains node y which has topology ID $id^{(y)}$, then $id^{(x)}$ and $id^{(y)}$ must satisfy the following two conditions:

(1) $id_0^{(x)} id_1^{(x)} \dots id_{i-1}^{(x)} = id_0^{(y)} id_1^{(y)} \dots id_{i-1}^{(y)} = d_0 d_1 \dots d_{i-1}$,

(2) $id_i^{(x)} \neq j$ and $id_i^{(y)} = j$.

Please note that there could exist a set Y of nodes meeting the above conditions for node x. For instance, table entry $T[3,5]$ in a node with octal topology ID 012345 can contain any node with a topology ID $\in 0125[0-7]^+$. Therefore we need a decision function to choose the best candidate. For instance, Plaxton chooses $y^* = Min_{y \in Y} CommCost(x,y)$ as the best neighbor.

In the case of $Y = \emptyset$, Plaxton assumes an ordering in the set of n nodes and picks a node y that matches node x in the suffix $i, i+1, \dots, r-1$ digits with the highest order. In contrast to the full Plaxton neighbor table, we leave those entries empty and prove this modification avoids cycles in the application-level multicasting (Theorem 7.2 in Sect. 7.8). In the application-level multicasting, very often a user program uses only a small portion of n processors. In an extreme case where a user program uses two out of n processors, our modification leads to routing tables with a unique non-empty entry while Plaxton neighbor table is full of the same process ID.

7.5.1 Compact routing table

While routing tables are typically used to connect nodes, we use them to connect processors (or processes) in the context of application-level multicasting. Assume a system has n processors and the base of topology IDs is equal to c, then the routing table will have $\frac{log_2(n)}{log_2(c)}$ rows and c columns. For instance, a system has 2048 processors. If we adopt a base of 8, the routing table has 4 rows and 8 columns, that is, 32 entries (shown in Fig. 7.6).

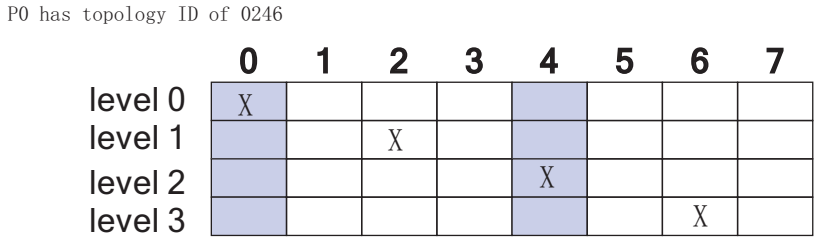


Figure 7.6: Routing table for a system with 2048 processors.

The routing table occupies a small amount of space even for large-scale systems. If a system has one million (2^{20}) cores, a base of 16 results in a routing table of 5 rows by 16 columns that equals 80 entries. If one has a billion (2^{30}) cores, the routing table is of 6 rows and 32 columns given the base 32. Every table entry just stores a single integer.

7.6 Algorithms

This section describes how each process builds its own routing table when the application first starts and how the process constantly receives messages and forwards them to proper destinations.

7.6.1 Building routing tables

Before doing any real work, each MPI process first builds a local routing table. Each process's topology ID is assigned by the user based on the network topology. In the program shown in Fig. 7.7, a process scans every other process ID and compares that process's topology ID to its own topology ID to fill in the routing table. When there are multiple processes that are legitimate to be stored in T[i,j], we either pick a process randomly or find the closest process. In our experiments on a Myrinet network, the random method is slightly better than the nearest neighbor method.

```
typedef struct {
  int table[NUM_LEVELS * NUM_COLS];
  int topo_ids[MAX_NUM_PROCESSES];
}* NeighborTable;

int*           candidates[NUM_LEVELS * NUM_COLS];
NeighborTable  my_tbl;

for(p = 0; p < nprocs; p++) {
  if(p  == my_pid) continue;
  topid  = my_tbl->topo_ids[p];
  level  = longest_comm_prefix(my_top_id, topid);
  column = get_kth_digit(topid, level);
  idx    = level * NUM_COLS + column;
  candidates[idx][counters[idx]++] = p;
}
/* Choose a proper neighbor from candidates */
choose_best_neighbor(my_tbl, candidates);
```

Figure 7.7: C program to build routing tables.

7.6.2 The forwarding algorithm

While participating in the multicast, a process works as either an internal node or a leaf in the multicast tree. Whenever a root is given, the locations of receivers become fixed in the particular multicast tree. Data will always flow from root to leaves. Figure 7.8 shows how to find the next level of tree nodes in the multicast tree to which to forward. The index of the next level should be at least one level further from the root. In the program shown in Fig. 7.8, `stage` indicates the index of the next level. The program looks up the table and gets a forwarding process for each child and stores it in array `destinations`.

7.7 Understanding how it works

The nonblocking multicast scheme is similar to the prefix-based routing method. Every node has a topology ID. When forwarding a message to a destination, the current node determines an intermediate node to forward the message to merely based on the destination's topology ID (i.e., a string or label). Every entry in the routing table works like a "channel" which

```
while(1) {
  ...
  Received a message from the process prev_topid;
  stage = longest_comm_prefix(my_top_id, prev_topid) + 1;
  for(i = 0; i < num_children; i++) {
    p = get_children(i);
    if(p == my_pid) continue;
    top_id = my_tbl->topo_ids[p];
    lcl = longest_comm_prefix(my_top_id, top_id);
    if(lcl  >= stage) {
      column  = get_kth_digit(top_id, lcl);
      forward = TBL_ENTRY(my_tbl, lcl, column);
      if(!is_element(forward, destinations)) {
        destinations[idx++] = forward;
      }
    }
  }
  Send the message to processes stored in destinations[];
}
```

Figure 7.8: C program to perform forwarding.

stands for a specific set of nodes. We could imagine that each "channel" is denoted by a regular expression. Figure 7.9 depicts a system with 8 nodes. By definition, the routing table for the system has 3 rows and 2 columns. Except for itself, every process stores only three neighbors. Process 001's routing table is shown on the right hand side of Fig. 7.9. The corresponding regular expressions of the three entries are: `1**`, `01*`, and `000`. Given any destination, process 001 compares the destination's topology ID to the three regular expressions to find the longest match and do the forwarding. It is easy to find that the three regular expressions partition the other 7 nodes on the system.

Another important characteristic of the nonblocking multicast scheme is that there are actually multiple spanning trees on the system. As shown in Fig. 7.10, there are two types of roots in the spanning tree for a 16-node system (base=2): `0***` and `1***`. Every node has a different spanning tree with itself as the root. For instance, the node `0000` belongs to the set `0***` and serves as the root of its own tree. Its neighbor on level 0 could be an arbitrary node that satisfies the regular expression of `1***`.

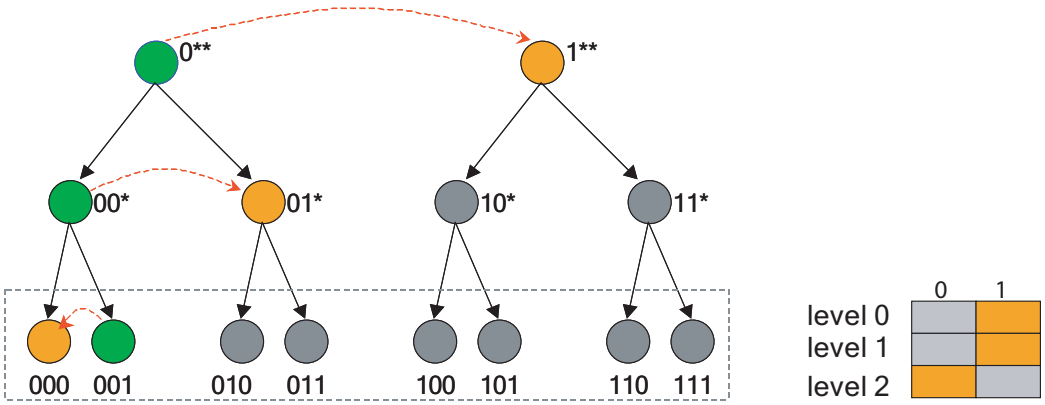


Figure 7.9: Similar to the prefix-based routing method.

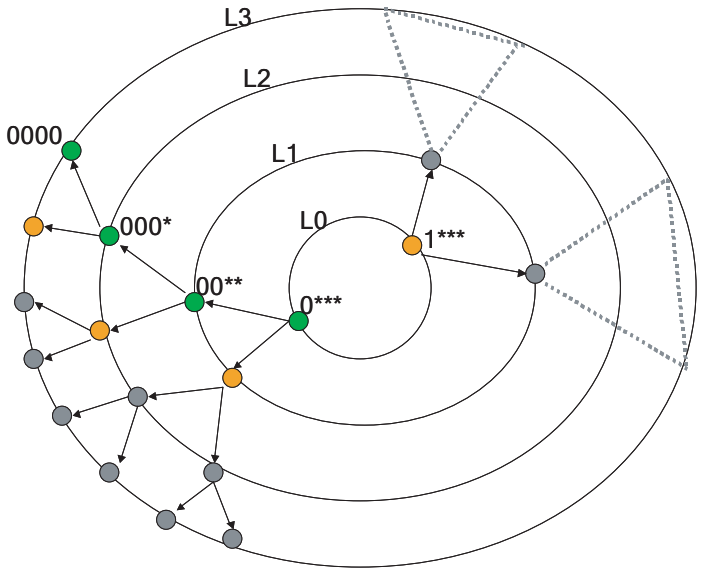


Figure 7.10: Implicit multiple spanning trees.

7.8 Theorems

Lemma 7.1. *Suppose process P_x has a topology ID x and needs to send data to process P_z with topology ID z. Then there always exists a process P_y stored in P_x's neighbor table such that P_x can forward data to P_y and $LCD(y,z) \geq LCD(x,z)+1$.*

Proof. Let $LCD(i,j)$ compute the longest common prefix length of i and j. Suppose $i = LCD(x,z)$, P_x will forward data to a process with a topology ID of the form $x_0x_1 \ldots x_{i-1}z_i* \ldots *$. It is easy to see that at least z has the form. So one of the processes of $P_z \cup \{\text{processes with ID } x_0x_1 \ldots x_{i-1}z_i * \ldots *\}$ will get the forwarded data. Therefore such a process must exist. By definition of LCD, we know $x_0x_1 \ldots x_{i-1} = z_0z_1 \ldots z_{i-1}$ and $x_i \neq z_i$. Given i and z, the forwaring algorithm chooses the process stored in the ith row and the z_ith column. Any process located in $T[i,z_i]$ will be the target to which P_x forwards data and it must exist. WLOG, let it be P_y with topology ID y. Since P_y is in $T[i,z_i]$ of P_x's neighbor table, $y_0y_1 \ldots y_{i-1} = x_0x_1 \ldots x_{i-1} = z_0z_1 \ldots z_{i-1}$ and $y_i = z_i$. Therefore, $y_0y_1 \ldots y_i = z_0z_1 \ldots z_i$. In other words, $LCD(y,z) \geq i+1 = LCD(x,z)+1$. □

Lemma 7.1 proves that the forwarding method is always successful even if there exist empty entries in the process's routing table. For every step of forwarding, the longest common prefix length to the destination increases by at least one.

Theorem 7.1 (Reachability)**.** *It is always possible to route a message from process P_x to process P_z and it takes at most m steps to reach P_z. m is the number of digits in topology IDs.*

Proof. By Lemma 7.1, there $\exists P_y$ such that P_x can forward data to P_y and $LCD(y,z) \geq LCD(x,z)+1$. Since topology ID z has m digits, it takes at most m steps to send data to P_z.

□

Theorem 7.2 (Deadlock-freedom)**.** *The forwarding mechanism guarantees that there is no cycle during the forwarding process.*

Proof. Suppose process x_1 wants to send a message to x_p, but there is a cycle $x_1 \rightarrow x_2 \ldots \rightarrow x_k \rightarrow x_1$ formed before the message reaches x_p. If $LCD(x_1, x_p) = d_0 d_1 \ldots d_{l_1}$, then by Lemma 7.1,

$$\begin{aligned}
LCD(x_2, x_p) &= d_0 d_1 \ldots d_{l_1} \ldots d_{l_2} \\
LCD(x_3, x_p) &= d_0 d_1 \ldots d_{l_1} \ldots d_{l_2} \ldots d_{l_3} \\
&\ldots \\
LCD(x_k, x_p) &= d_0 d_1 \ldots d_{l_1} \ldots d_{l_2} \ldots d_{l_3} \ldots d_{l_k} \\
LCD(x_1, x_p) &= d_0 d_1 \ldots d_{l_1} \ldots d_{l_2} \ldots d_{l_3} \ldots d_{l_k} \ldots d_{l_{k+1}}
\end{aligned}$$

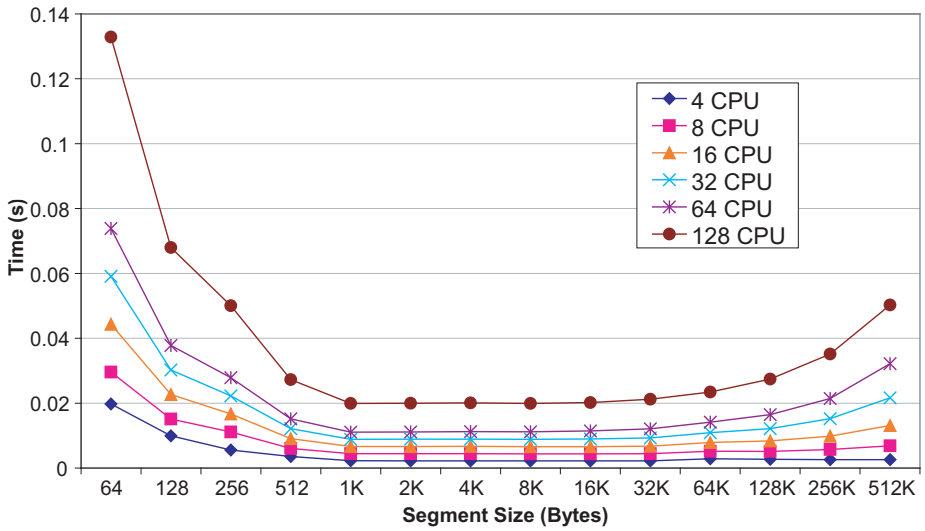
There is a contradiction if we compare the first $LCD(x_1, x_p)$ and the last $LCD(x_1, x_p)$. Therefore, the forwarding mechanism guarantees there is no cycle. □

7.9 Experiments

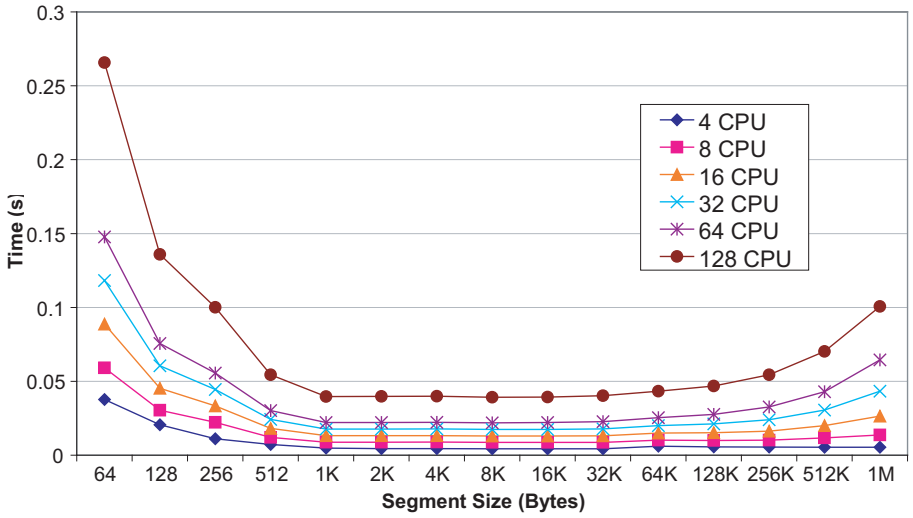
We conducted experiments on a cluster machine with 64 nodes each with two processors. The cluster is connected by a Myrinet network. We also did experiments on a SGI Altix 3700 BX2 machine which has a fat tree network topology. The SGI Altix machine is connected by a NUMAlink 4 network.

7.9.1 Effect of segment size

The performance of the non-blocking multicast method could be affected by the segment size. Given a message size, we can choose to send it out once or in a number of segments. We consider two message sizes: 512KB and 1MB. For each message size, we use different segments with sizes from 64Bytes to the whole message size and run it on a range of processors from 4 CPUs to 128 CPUs. Figures 7.11 and 7.12 show the effect of segment size on the Myrinet network and SGI's NUMAlink 4 network, respectively.

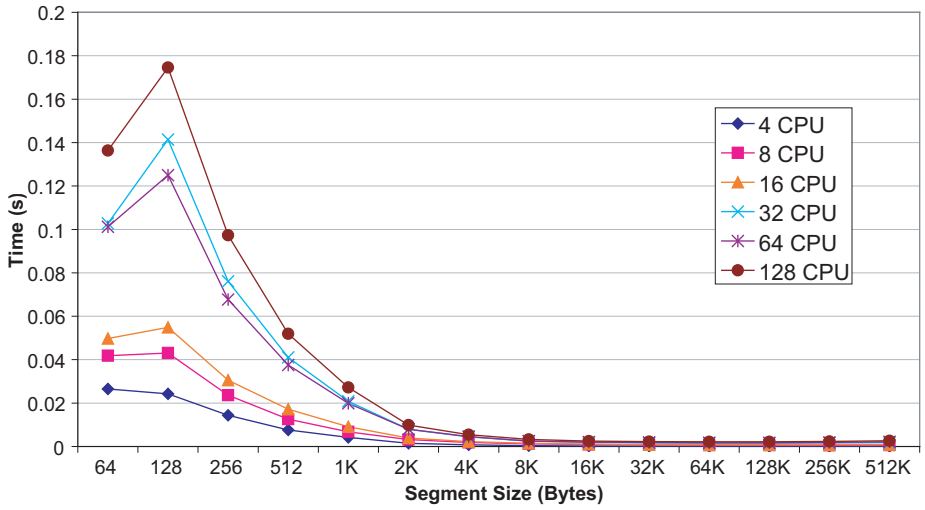


(a) Multicast a 512KB message.

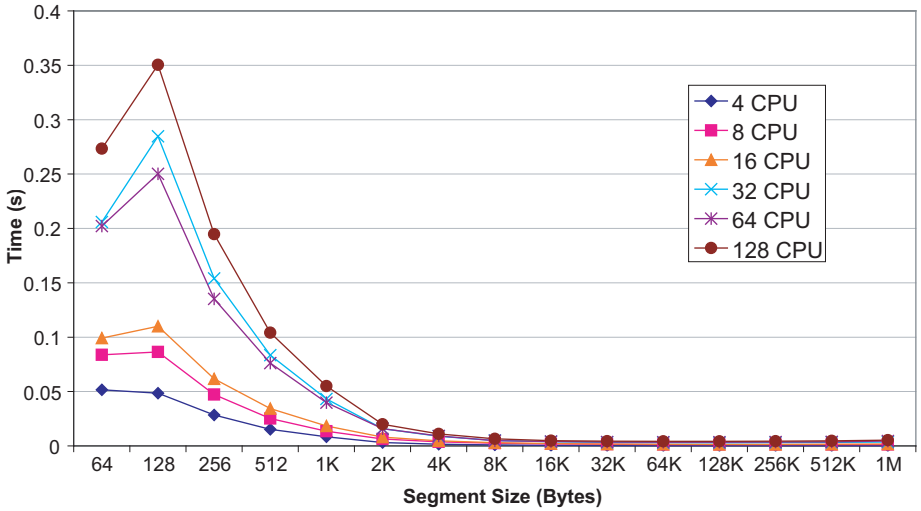


(b) Multicast a 1MB message.

Figure 7.11: The multicast performance varies with different segment sizes on Myrinet.



(a) Multicast a 512KB message.



(b) Multicast a 1MB message.

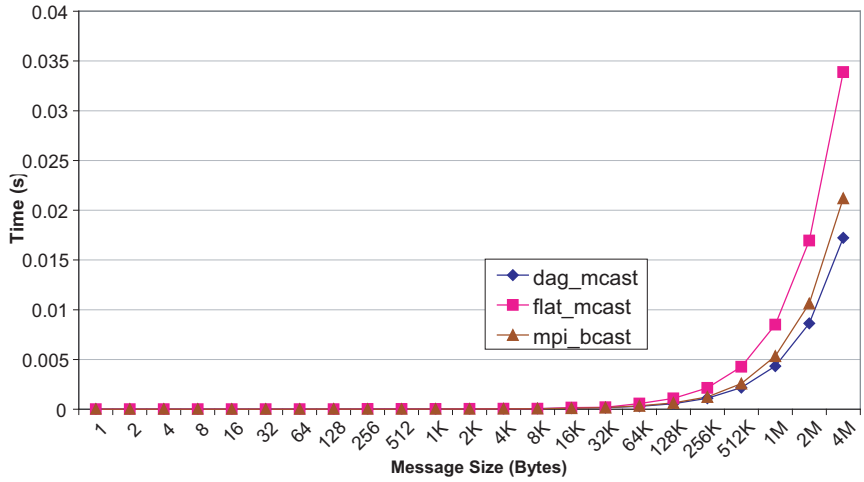
Figure 7.12: The multicast performance varies with different segment sizes on SGI's NUMAlink.

Based on the data, a segment size between 1KB and 32KB always produces the best performance on Myrinet. And on SGI's NUMAlink, any segment of size $\geq$ 8KB can produce a good performance. Therefore in all the experiments described in Sect. 7.9.2, we choose the segment size of 8KB to do multicasting whenever possible.

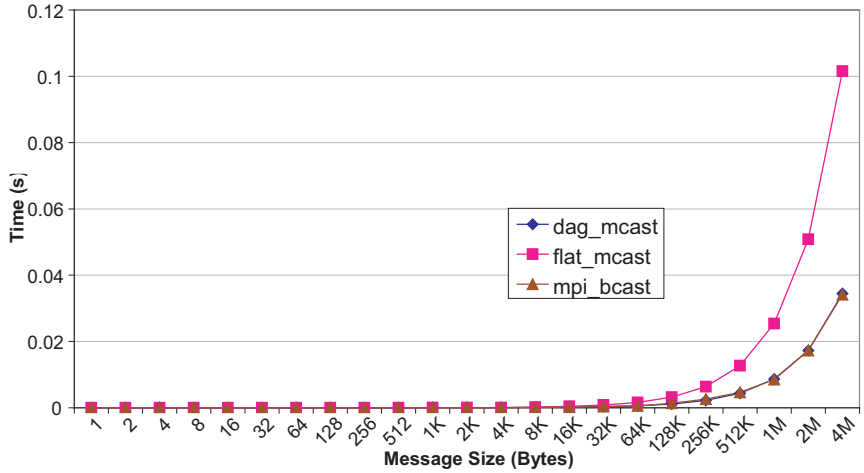
7.9.2 Experimental results

We compare our non-blocking multicast method (labeled as "dag_mcast") to MPI_Bcast (labeled as "mpi_bcast") and a straightforward implementation that uses a flat-tree to perform multicasting (labeled as "flat_mcast"). The flat-tree method simply sends the message to every destination one by one. Both flat_mcast and dag_mcast are implemented using the point-to-point `MPI_Send` and `MPI_Recv` operations. On the Myrinet cluster, we use the MPI library of `mpich-mx 1.0`. And on the SGI Altix machine, we use the SGI MPI library.

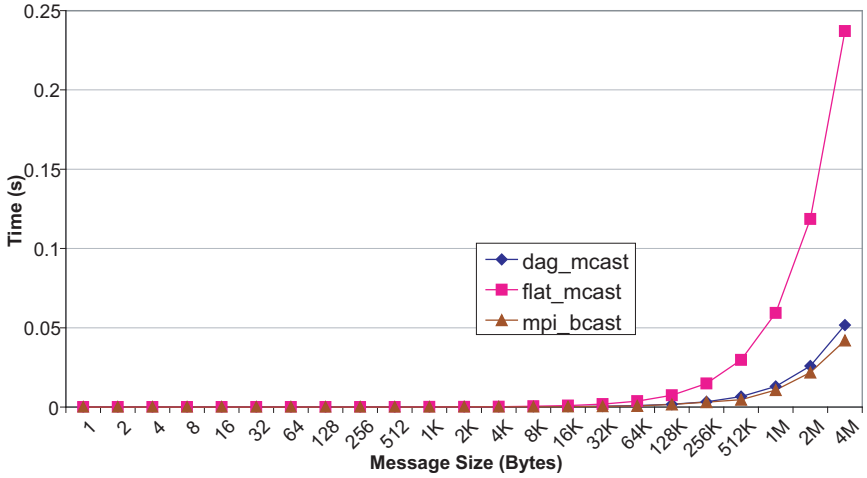
We conducted experiments on a range of processors from 4 up to 128. From Figs. 7.13 and 7.14 for Myrinet, and Figs. 7.15 and 7.16 for SGI NUMALink, we can see that both `dag_mcast` and `mpi_bcast` are significantly better than `flat_mcast`. And the non-blocking multicast method is comparable to the highly-optimized collective MPI_Bcast. Note that the time to invoke `MPI_Init` and `MPI_Comm_create` is not counted for the `mpi_bcast` experiments (in favor of `mpi_bcast`). The reason why the non-blocking multicast method is slower than MPI_Bcast is because our implementation is built over MPI point-to-point operations and we cannot do similar optimizations as MPI collective operations do (e.g., broadcast may be implemented as `scatter` followed by `allgather`, optimal binomial tree is built in advance). Although MPI_Bcast is faster, it is still very difficult to create communication groups and do collective broadcasts for every distinct group in dynamic DAG scheduling programs.



(a) 4 Processes

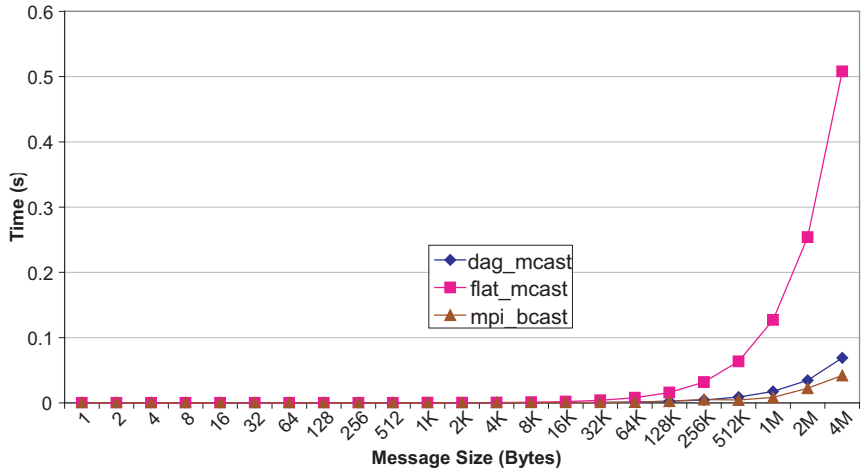


(b) 8 Processes

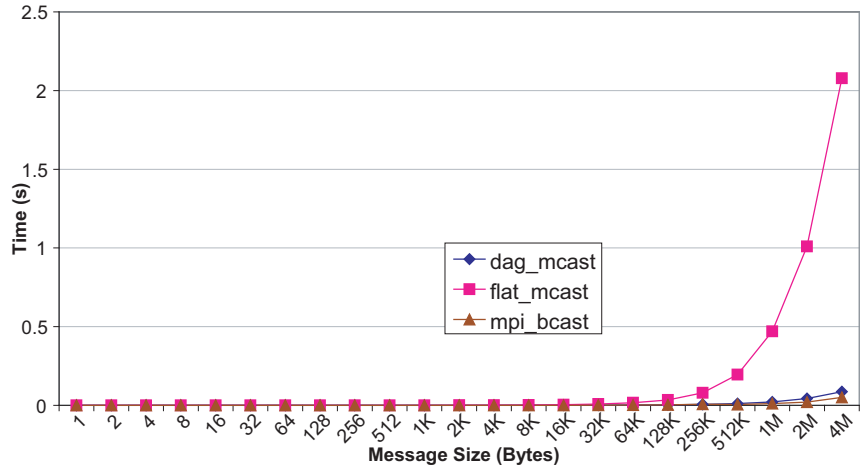


(c) 16 Processes

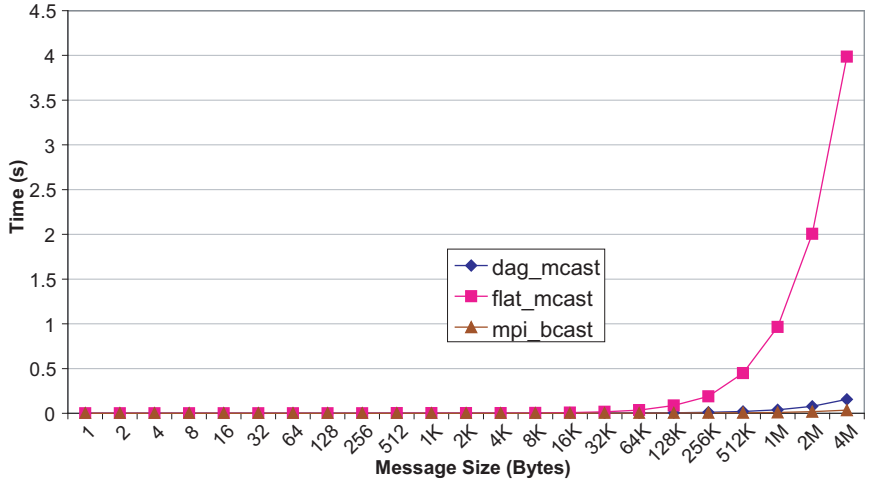
Figure 7.13: Multicast performance on a cluster connected with Myrinet (4-16 processes).



(d) 32 Processes

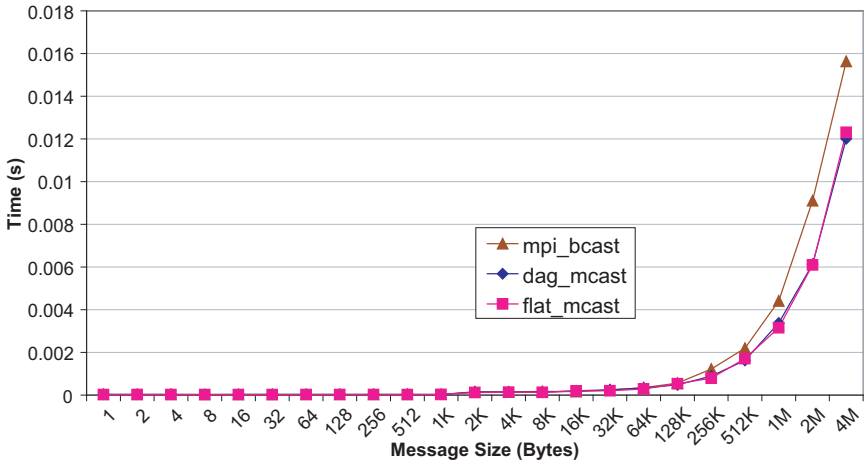


(e) 64 Processes

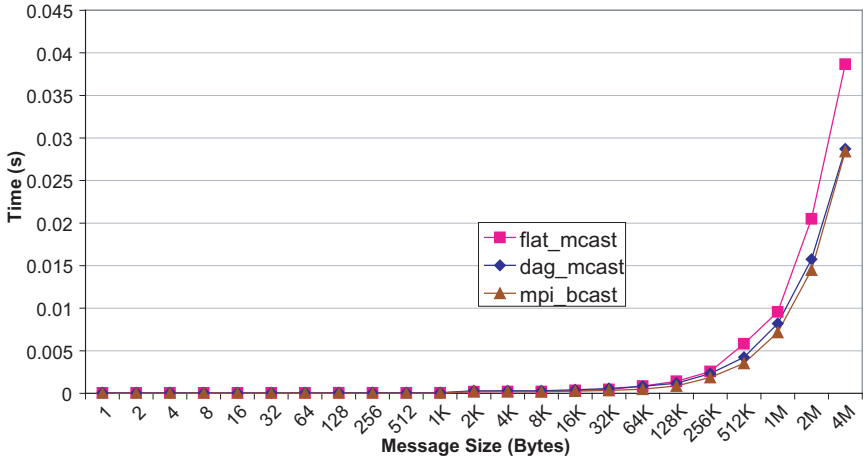


(f) 128 Processes

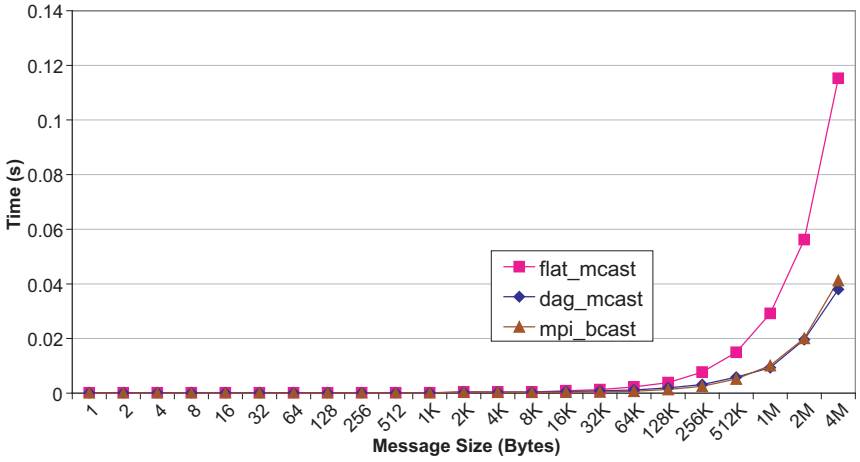
Figure 7.14: Multicast performance on a cluster connected with Myrinet (32-128 processes).



(a) 4 Processes

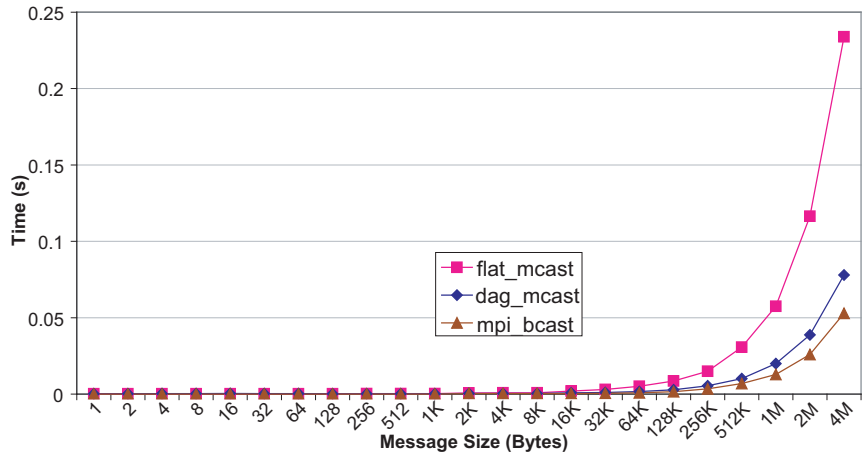


(b) 8 Processes

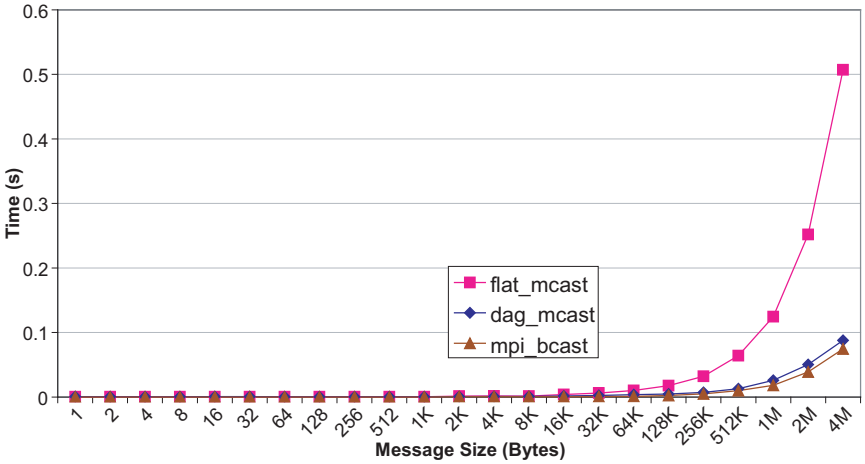


(c) 16 Processes

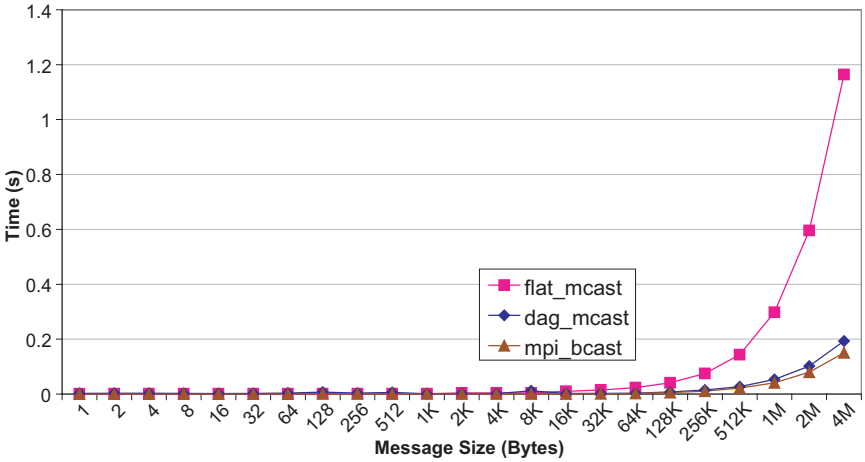
Figure 7.15: Multicast performance on SGI NUMAlink (4-16 processes).



(d) 32 Processes



(e) 64 Processes



(f) 128 Processes

Figure 7.16: Multicast performance on SGI NUMAlink (32-128 processes).

7.10 Summary

Our non-blocking multicast scheme is designed to support dynamic DAG scheduling on distributed-memory machines. While it is possible to use `MPI_Bcast` directly to implement it, creating communication groups and performing collective operations for arbitrary sets of parent/children is cumbersome to program. We have designed a multicast scheme, using topology IDs, compact routing tables, and multiple spanning trees. The multicast scheme is proven to be deadlock free, scalable in terms of time and space, topology-aware, and non-blocking. Our experimental results on a Myrinet network and SGI's NUMAlink show that our multicast scheme is significantly better than the simple flat-tree method and comparable to vendor-optimized collective MPI operations.

Chapter 8

Conclusions and Future Work

The multicore architecture raises a lot of challenging research problems regarding how to use the new architecture effectively. This dissertation starts with looking at the novel feature of the shared L2 cache on a single multicore processor to study the effect of cache sharing and contention. Inspired by the analytical cache model and the increasingly deeper memory hierarchy on multicore systems, we propose to use the affinity based thread scheduling to maximize the memory effectiveness on all the levels in a complex memory hierarchy. Multicore systems include not only shared-memory but also distributed-memory machines. Thus the dissertation also studies the dynamic data-availability driven scheduling approach to designing new parallel software on distributed-memory multicore systems.

8.1 Conclusions

In this dissertation, we first develop an analytical model to predict the number of cache misses on the shared L2 cache. The shared L2 cache may reduce the number of cache misses if the data are accessed in common by several threads, but it may also result in performance degradation due to resource contention. We use the circular sequence profiling and stack processing techniques to analyze the L2 cache trace to predict the number of compulsory cache misses, capacity cache misses on shared data, and capacity cache misses on private data, respectively. The model is able to predict the L2 cache performance for threads that

have a global shared address space. We use the cycle accurate simulator SESC to validate the model with three scientific programs: dense matrix multiplication, blocked dense matrix multiplication, and sparse matrix-vector product. The average relative errors for the three experiments are 8.01%, 1.85%, and 2.41%, respectively.

To investigate the affinity based thread scheduling, we have proposed an analytical model to estimate the cost of running an affinity based thread schedule on shared-memory multicore systems. The model consists of three submodels to evaluate the cost of executing a thread schedule: an affinity graph submodel, a memory hierarchy submodel, and a cost submodel that characterize programs, machines, and costs respectively. We apply the analytical model to both synthetic and real-world applications. The estimated cost accurately predicts which thread schedule will provide better performance. With the aid of the model, we are able to formulate the problem of determining the best thread schedule as an optimization problem. Due to the NP-hardness of the scheduling problem, we design an approximation algorithm to compute near-optimal solutions. We also extend the algorithm to support threads with data dependences. All the algorithms have been implemented in a feedback-directed optimization framework and applied to a number of scientific applications. Our experimental results show that using the optimized thread schedule can improve the program performance greatly (by up to 400%).

The dissertation finally uses a dynamic task scheduling approach to designing new linear algebra software on distributed-memory multicore systems. We use a task-based library to replace the existing linear algebra subroutines such as PBLAS to transparently provide the same interface and compute function as the ScaLAPACK library. We focus our runtime system design on the performance scalability metric. At any time the runtime system keeps a small portion of a task graph in memory. The runtime system distributes both data and task graphs across different compute nodes to achieve scalability. We propose an algorithm to solve data dependences without process cooperation in a distributed way. The runtime system distinguishes thread roles of task generation, task computing, and data communication. We have implemented the runtime system and applied it to three linear algebra algorithms: Cholesky factorization, LU factorization, and QR factorization. Our

experiments on shared-memory machines (16-core Intel Tigerton, 32-core IBM Power6) and distributed-memory machines (Cray XT4 using 1024 cores) demonstrate that our runtime system is able to achieve good scalability. We have provided analytical analysis to show why the tiled algorithms are scalable and the expected execution time. Furthermore, a non-blocking multicast scheme is introduced for dynamic DAG scheduling applications on distributed-memory systems. The multicast scheme takes into account both network topology and the space requirement of routing tables to achieve high scalability. We have proved that the scheme is deadlock-free and takes at most $logN$ steps to complete. Although built on MPI point-to-point operations, our experimental results show that the multicast scheme is close to the vendor optimized collective MPI operations.

8.2 Future work

The dissertation mainly focuses on the static affinity thread scheduling and the dynamic DAG scheduling research. Along the dissertation line, further research and improvement are possible.

The affinity graph submodel could be more accurate. On a cache-coherent shared memory system, a memory write often invalidates the corresponding cache lines in the other processors. The cache invalidation is an expensive operation. Therefore when partitioning affinity graphs, we can give a higher priority to writes than to reads. The submodel represents the affinity relationship by the number of addresses accessed in common by threads. A more accurate way would be to use cache lines (in virtual memory addresses) to reflect the actual data movement between CPUs and caches such as load/store of cache lines. It can also reduce the space complexity of the trace analysis algorithms.

The feedback directed optimization prototype system is limited by the total amount of memory due to its diskless scheme. A large graph with hundreds of millions of vertices that has no edge alone still requires too much memory. We can revise our algorithms to create and store graphs to disks to solve the issue. We can even design parallel distributed algorithms to make use of more memories. It would also be possible to implement the feedback-directed

strategy in commercial compilers so that the programmer can automatically achieve high-performance on leading-edge shared memory systems.

In the dynamic DAG scheduling approach, a large block size NB is desirable because it can maximize the cache hit rate and compensate for the associated scheduling overhead. But when the input matrix is small, we must choose a small block size to keep the parallelism degree sufficiently high. For small blocks (or vectors), we need to add an affinity-based mechanism to our scheduling runtime system to improve data reuse between tasks. We also need to continue to optimize our runtime system to minimize the scheduling expense. Improving data reuse is even more critical on NUMA multicore systems. So far we assume that processor cores are homogeneous and have the same computational power. An important research would be to design a scalable dynamic runtime system on heterogeneous architectures such as hybrid CPU+GPU manycore machines.

Bibliography

Bibliography

[Agarwal et al., 1989] Agarwal, A., Horowitz, M., and Hennessy, J. (1989). An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215.

[Asanovic et al., 2006] 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. (2006). The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[Ayguade et al., 2009] Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of OpenMP tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418.

[Balakrishnan et al., 2005] Balakrishnan, S., Rajwar, R., Upton, M., and Lai, K. K. (2005). The impact of performance asymmetry in emerging multicore architectures. In *32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA*, pages 506–517. IEEE Computer Society.

[Banerjee et al., 2002] Banerjee, S., Bhattacharjee, B., and Kommareddy, C. (2002). Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217, New York, NY, USA. ACM.

[Blackford et al., 1996] Blackford, L. S., Choi, J., Cleary, A., Petitet, A., Whaley, R. C., Demmel, J., Dhillon, I., Stanley, K., Dongarra, J., Hammarling, S., Henry, G., and

Walker, D. (1996). ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 5, Washington, DC, USA. IEEE Computer Society.

[Blumofe and Leiserson, 1999] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748.

[Boisvert et al., 1996] Boisvert, R., Pozo, R., Remington, K., Barrett, R., and Dongarra, J. (1996). Matrix Market: a web resource for test matrix collections. In *Quality of Numerical Software*, pages 125–137.

[Buttari et al., 2009] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53.

[Chandra et al., 2005] Chandra, D., Guo, F., Kim, S., and Solihin, Y. (2005). Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005*, pages 340–351.

[Choi et al., 1996] Choi, J., Dongarra, J. J., Ostrouchov, L. S., Petitet, A. P., Walker, D. W., and Whaley, R. C. (1996). Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Sci. Program.*, 5(3):173–184.

[Cosnard and Jeannot, 1999] Cosnard, M. and Jeannot, E. (1999). Compact DAG representation and its dynamic scheduling. *J. Parallel Distrib. Comput.*, 58(3):487–514.

[Davis, 1997] Davis, T. A. (1997). University of Florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse.

[Ding and Orlovich, 2004] Ding, C. and Orlovich, M. (2004). The potential of computation regrouping for improving locality. In *Proceedings of the ACM/IEEE SC2004 Conference on High Performance Networking and Computing, 6-12 November 2004, Pittsburgh, PA, USA, CD-Rom*, page 13. IEEE Computer Society.

[Dongarra et al., 1992] Dongarra, J., van de Geijn, R., and Walker, D. (1992). A look at scalable dense linear algebra libraries. *Scalable High Performance Computing Conference. SHPCC-92. Proceedings.*, pages 372–379.

[El-Rewini et al., 1994] El-Rewini, H., Lewis, T. G., and Ali, H. H. (1994). *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Fedorova et al., 2006] Fedorova, A., Seltzer, M., and Smith, M. (2006). A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*.

[Frigo et al., 1998] Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA. ACM.

[Golla, 2007] Golla, R. (2007). Niagara2: A highly threaded server-on-a-chip.

[Hammond et al., 1997] Hammond, L., Nayfeh, B. A., and Olukotun, K. (1997). A single-chip multiprocessor. *Computer*, 30(9):79–85.

[Hendrickson and Womble, 1994] Hendrickson, B. A. and Womble, D. E. (1994). The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Comput.*, 15(5):1201–1226.

[Hill and Smith, 1989] Hill, M. and Smith, A. (1989). Evaluating associativity in CPU caches. *IEEE Trans. Computers*, 38(12):1612–1630.

[Karonis et al., 2003] Karonis, N. T., Toonen, B., and Foster, I. (2003). MPICH-G2: A Grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551 – 563. Special Issue on Computational Grids.

[Kumar et al., 2006] Kumar, R., Tullsen, D. M., and Jouppi, N. P. (2006). Core architecture optimization for heterogeneous chip multiprocessors. In Altman, E. R., Skadron, K., and

Zorn, B. G., editors, *15th International Conference on Parallel Architecture and Compilation Techniques (PACT 2006), Seattle, Washington, USA, September 16-20, 2006*, pages 23–32. ACM.

[Kumar et al., 2004] Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. (2004). Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*, pages 64–75. IEEE Computer Society.

[Kurzak et al., 2008] Kurzak, J., Buttari, A., and Dongarra, J. (2008). Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Trans. Parallel Distrib. Syst.*, 19(9):1175–1186.

[Le et al., 2007] Le, H. Q., Starke, W. J., Fields, J. S., O'Connell, F. P., Nguyen, D. Q., Ronchetti, B. J., Sauer, W. M., Schwarz, E. M., and Vaden, M. T. (2007). IBM Power6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662.

[Lu et al., 1995] Lu, H., Dwarkadas, S., Cox, A. L., and Zwaenepoel, W. (1995). Message passing versus distributed shared memory on networks of workstations. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 37. ACM Press.

[Luk et al., 2005] Luk, C.-K., Cohn, R. S., Muth, R., Patil, H., Klauser, A., Lowney, P. G., Wallace, S., Reddi, V. J., and Hazelwood, K. M. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200.

[Markatos and LeBlanc, 1994] Markatos, E. P. and LeBlanc, T. J. (1994). Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 5(4):379–400.

[Mattson et al., 1970] Mattson, R., Gecsei, J., Slutz, D., and Traiger, I. (1970). Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117.

[Panda et al., 1999] Panda, D., Singal, S., and Kesavan, R. (1999). Multidestination message passing in wormhole k-ary n-cube networks with base routing conformed paths. *Parallel and Distributed Systems, IEEE Transactions on*, 10(1):76–96.

[Park et al., 2003] Park, N., Hong, B., and Prasanna, V. K. (2003). Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654.

[Perez et al., 2008] Perez, J., Badia, R., and Labarta, J. (2008). A dependency-aware task-based programming environment for multi-core architectures. *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151.

[Philbin et al., 1996] Philbin, J., Edler, J., Anshus, O. J., Douglas, C. C., and Li, K. (1996). Thread scheduling for cache locality. In *ASPLOS*, pages 60–71.

[Pichel et al., 2004] Pichel, J. C., Heras, D. B., Cabaleiro, J. C., and Rivera, F. F. (2004). Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. In *12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2004), 11-13 February 2004, A Coruna, Spain*, pages 66–71. IEEE Computer Society.

[Pichel et al., 2005] Pichel, J. C., Heras, D. B., Cabaleiro, J. C., and Rivera, F. F. (2005). A new technique to reduce false sharing in parallel irregular codes based on distance functions. In *8th International Symposium on Parallel Architectures, Algorithms, and Networks, ISPAN 2005, December 7-9. 2005, Las Vegas, Nevada, USA*, pages 306–311. IEEE Computer Society.

[Pingali et al., 2003] Pingali, V. K., McKee, S. A., Hsieh, W. C., and Carter, J. B. (2003). Restructuring computations for temporal data cache locality. *International Journal of Parallel Programming*, 31(4):305–338.

[Plaxton et al., 1997] Plaxton, C. G., Rajaraman, R., and Richa, A. W. (1997). Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA. ACM.

[Reinders, 2007] Reinders, J. (2007). *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[Renau et al., 2005] Renau, J., Fraguela, B., Tuck, J., Liu, W., and Prvulovic, M. (2005). SESC simulator. http://sesc.sourceforge.net.

[Seiler et al., 2008] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. (2008). Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15.

[SGI, 2006] SGI (2006). Linux resource administration guide. In *SGI Techpubs Library 007-4413-011*.

[Simon and Teng, 1997] Simon, H. D. and Teng, S.-H. (1997). How good is recursive bisection? *SIAM J. Sci. Comput.*, 18(5):1436–1445.

[Song et al., 2009a] Song, F., Dongarra, J., and Moore, S. (2009a). A scalable non-blocking multicast scheme for distributed DAG scheduling. In Allen, G., Nabrzyski, J., Seidel, E., van Albada, G. D., Dongarra, J., and Sloot, P. M. A., editors, *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*, volume 5544 of *Lecture Notes in Computer Science*, pages 195–204. Springer.

[Song et al., 2007a] Song, F., Moore, S., and Dongarra, J. (2007a). Feedback-directed thread scheduling with memory considerations. In Kesselman, C., Dongarra, J., and

Walker, D. W., editors, *Proceedings of the 16th International Symposium on High-Performance Distributed Computing (HPDC-16 2007), 25-29 June 2007, Monterey, California, USA*, pages 97–106. ACM.

[Song et al., 2007b] Song, F., Moore, S., and Dongarra, J. (2007b). L2 cache modeling for scientific applications on Chip Multi-Processors. In *2007 International Conference on Parallel Processing (ICPP 2007), September 10-14, 2007, Xi-An, China*, pages 51–58. IEEE Computer Society.

[Song et al., 2009b] Song, F., Moore, S., and Dongarra, J. (2009b). Analytical modeling and optimization for affinity based thread scheduling on multicore platforms. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing, 31 August - 4 September 2009, New Orleans, Louisiana, USA*. IEEE.

[Song et al., 2009c] Song, F., YarKhan, A., and Dongarra, J. (2009c). Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. IEEE/ACM.

[Suh et al., 2002] Suh, G., Devadas, S., and Rudolph, L. (2002). A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 117–128.

[Thiébaut and Stone, 1987] Thiébaut, D. and Stone, H. (1987). Footprints in the cache. *ACM Trans. Comput. Syst.*, 5(4):305–329.

[Träff, 2002] Träff, J. L. (2002). Implementing the MPI process topology mechanism. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Wu and Sheng, 2005] Wu, J. and Sheng, L. (2005). Deadlock-free multicasting in irregular networks using prefix routing. *The Journal of Supercomputing*, 31:63–78.

[Yan et al., 2000] Yan, Y., Zhang, X., and Zhang, Z. (2000). Cacheminer: A runtime approach to exploit cache locality on SMP. *IEEE Trans. Parallel Distrib. Syst.*, 11(4):357–374.

[Zhao et al., 2001] Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, Berkeley, CA, USA.

[Zhuang et al., 2001] Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R. H., and Kubiatowicz, J. D. (2001). Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA. ACM.

Vita

Fengguang Song was born in Hebi, Henan, China, on April 26, 1974. He received his Bachelor of Engineering in Computer Science in 1996 from Zhengzhou University and Master of Engineering in Computer Science in 1999 from Nanjing University of Aeronautics and Astronautics, respectively. Between 2000 and 2002, he studied in the Department of Computer Science at the University of British Columbia and earned his Master of Science degree. In 2003 he was enrolled in the University of Tennessee as a doctoral student in Computer Science. He joined the Innovative Computing Laboratory as a graduate research assistant in May, 2003 where he completed his Master of Science degree in 2008 and Doctor of Philosophy degree in 2009.