Efficient Irregular Computation on High-Bandwidth Pipelined-Memory Multiprocessors

Marco Zagha

May 1998

CMU-CS-98-128

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted to Carnegie Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Thesis Committee:
Guy E. Blelloch, Chair
Allan L. Fisher
Phillip B. Gibbons, Bell Laboratories
David R. O'Hallaron

Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Defense Advanced Research Projects Agency (DARPA) under grant number F33615-93-1-1330. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation therein. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the U.S. Government. CRAY C90 and CRAY J90 time was provided by the Pittsburgh Supercomputing Center.

Keywords: parallel algorithms, latency hiding, PRAM, QRQW, BSP, CRAY C90, CRAY J90, vector multiprocessor, memory contention, interleaved memories, memory banks, benchmarks, sorting, merging, sparse matrices, graph algorithms

Abstract

A major goal in the parallel computing community has been to transfer results from the theory of parallel computing to the practice of developing efficient implementations of general-purpose parallel algorithms. However, the most widely-used theoretical model, the Parallel Random Access Machine (PRAM), is an impractical model of real parallel machines. Thus, much current research centers on redesigning algorithms for models that account for the communication bottlenecks present on most distributed memory machines, a task that is feasible for certain classes of algorithms, but is fundamentally difficult for algorithms that use irregular data structures and access patterns.

Instead, this thesis considers whether it is feasible to map PRAM algorithms onto a class of realistic parallel machines with only a subset of the features of an ideal PRAM. Our goal is to show that general-purpose PRAM algorithms can be implemented efficiently and systematically on a class of high-bandwidth shared-memory parallel machines that

1. have a high-bandwidth network between the processors and memory banks,
2. allow for fine-grained memory accesses,
3. can tolerate latency to the memory from processors by allowing for multiple outstanding memory requests, and
4. have memory banks that are slower than the processors and compensate by having more memory banks than processors.

These features are currently provided on vector multiprocessors, and might be provided in future parallel architectures. To model these machines, we extend Valiant's Bulk Synchronous Parallel (BSP) model to include machines that compensate for slow memory banks by having more memory banks than processors. We evaluate this extended model on the CRAY C90 and CRAY J90 and explore experimentally how various PRAM variants can be emulated in this model, with particular emphasis on the Queued-Read Queued-Write PRAM.

The major focus of the thesis is a practical study of techniques for implementing and modeling irregular PRAM algorithms on vector multiprocessors. The thesis describes a set of case studies on fundamental PRAM algorithms, including sorting, merging, sparse matrix multiplication, and graph connectivity. Each case study describes the selection of a practical PRAM algorithm, the process of mapping the algorithm onto the machine model, a performance model for predicting running time, and practical issues and tradeoffs in developing fast, general-purpose implementations for the CRAY C90 and CRAY J90.

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
COMPUTER SCIENCE

Efficient Irregular Computation on High-Bandwidth Pipelined-Memory Multiprocessors

MARCO ZAGHA

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

ACCEPTED:

[illegible] THESIS COMMITTEE CHAIR	May 14, 1998 DATE
[illegible] DEPARTMENT HEAD	5/14/98 DATE

APPROVED:

[illegible] DEAN	5/14/98 DATE

Acknowledgments

I would first like to express my deepest appreciation to my advisor, Guy Blelloch, for his invaluable technical guidance and tireless moral support. He was instrumental in helping me to formulate and focus my ideas—and in convincing me, in the age of MPPs, that "PRAM" and "vector" weren't dirty words. I would also like to thank the other members of thesis my committee—Allan Fisher, Phil Gibbons, and Dave O'Hallaron—for their helpful suggestions.

Portions of this thesis are based on joint work: Chapter 2 on modeling pipelined-memory multiprocessors with Guy Blelloch, Phil Gibbons (Bell Labs), and Yossi Matias (Bell Labs); Chapter 4 on sorting with Guy Blelloch; and Chapter 6 with Guy Blelloch and Mike Heroux (Cray Research). John Greiner provided valuable input on graph connectivity algorithms discussed in Chapter 7. Charles Grassl and Mike Heroux at Cray Research provided valuable technical assistance and performance data.

It was a pleasure working on some "dengerous" research with the members of the Scandal project—Sid Chatterjee, John Greiner, Jonathan Hardwick, Girija Narlikar, Margaret Reid-Miller, and Jay Sipelstein—as well as fellow NESL enthusiasts Keith Gremban, Gary Miller, Tom Sheffler, and Dafna Talmor. Special thanks to Jonathan Hardwick for suffering through early drafts of the thesis.

Summers at Thinking Machines were inspirational and educational. It was a privilege to work with many talented people, especially Charles Leiserson, Steve Heller, and the rest of the "supersort" gang, who taught me a lot about bringing together theory and practice—not to mention how to write "way fast" code.

I am indebted to Woody Lichtenstein and Horst Simon for helping me get a "real job" at Silicon Graphics, and to Jeff McDonald at SGI for his support and encouragement while I was finishing my thesis.

The School of Computer Science was a incredible place to work with an incredible bunch of people. Special thanks to Sharon Burks and Catherine Copetas for all that they do. Over the (many) years, I had a lot of great officemates: Erik Brunvand, Alex Hauptmann, Ellen Siegel, Rick Lerner, Erik Seligman, Su-Yuen Ling, Sean Smith, Srinivas Akella, Fred Solomon, Dafna Talmor, Chris McConnell, and "Rex" Xi Xu. And I can't forget my temporary officemate Marko Petkovsek, who briefly gave our office the distinction of having "more Marco's than women". I enjoyed the company of many friends, particularly Robert Cohn, Steve Schlick, Ven Seshadri, Bruce Siegell, and all the folks on the "dinner instance".

I can't forget my canine friend, Wooster, who deserves special thanks for helping me keep my sanity. And saving the most important acknowledgement for last, I give my deepest thanks to my wife Kim for her never-ending encouragement, patience, and loving support. I could not have done this without her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A major goal in the parallel computing community has been to transfer results from the theory of parallel computing to the practice of developing efficient implementations of general-purpose parallel algorithms. However, the most widely-used theoretical model, the Parallel Random Access Machine (PRAM) [112, 166, 117], is an impractical model of real parallel machines because it assumes a number of idealistic characteristics, including a high bandwidth global memory, and no penalties for latency, communication overhead, and synchronization. Some researchers view the discrepancy between the PRAM model and current parallel machines as an incentive to discard the PRAM model and to redesign algorithms for a model that accounts for limited communication (e.g., LOGP [62]).

Redesigning algorithms is feasible for certain classes of algorithms, but is fundamentally difficult for algorithms that use irregular data structures and access patterns. Thus, we reexamine the question of whether the PRAM could be used effectively in practice. We consider whether it is feasible to map PRAM algorithms onto a class of realistic parallel machines with only a subset of the features of an ideal PRAM. In particular we consider a class of shared-memory machines that

1. have a high-bandwidth network between the processors and memory banks,
2. allow for fine-grained memory accesses,
3. can tolerate latency to the memory from processors by allowing for multiple outstanding memory requests, and
4. have memory banks that are slower than the processors and compensate by having more memory banks than processors.

These features are currently provided on vector multiprocessors, and might be provided in future parallel architectures. We refer to a machine with these features as a high-bandwidth pipelined-memory multiprocessor (or simply a high-bandwidth multiprocessor). There are three main reasons for assuming high bandwidth:

performance: to use architectures that provide highest absolute performance on irregular algorithms, including algorithms that use frequent pointer chasing and permutation. Analysis of industry-standard benchmarks suggests that there is a strong correlation between global bandwidth and performance, particularly for irregular algorithms [35].

generality: to use general-purpose algorithms, rather than specializing algorithms to accommodate limited communication. For example, we want to sort arbitrary distributions of keys, and label connected components of arbitrary irregular graphs.

algorithm design effort: to explore the limits of what is possible using PRAM algorithms without expending the programming effort to redesign algorithms [35]. Instead, we focus on systematic implementation of PRAM algorithms on high-bandwidth multiprocessors.

Our thesis is that practical, irregular, PRAM algorithms can be used for implementing high-performance irregular computation on high-bandwidth multiprocessors. By "practical", we mean that the algorithms have small constant factors in their running time for problem sizes and machine sizes of practical interest. By "irregular", we mean algorithms that require a large amount of input-dependent information to describe their memory reference patterns, for example, algorithms using pointer-based data structures such as graphs and trees. By "PRAM algorithms", we mean that the algorithms assume a high-throughput fine-grained shared memory. (However, implementations of the algorithms do not assume unit-time latency.) For several irregular problems, our approach to programming high-bandwidth vector multiprocessors using PRAM algorithms has lead to the fastest implementations on vector machines, and the fastest general-purpose implementations for any machine.

In exploring this thesis, we use an experimental approach toward addressing several research questions:

- What is an appropriate machine model for high-bandwidth multiprocessors?
- What is an appropriate high-level programming model that balances accuracy, simplicity, and generality?
- What techniques are the most effective for developing fast general-purpose implementations of irregular PRAM algorithms?

The thesis first introduces a machine model for high-bandwidth pipelined-memory multiprocessors. The model is an extension of Valiant's Bulk Synchronous Parallel (BSP) [201, 181] model that includes machines that compensate for slow memory banks by having more memory banks than processors. (Although we have chosen to extend the BSP model, it should be straightforward to extend other related models, e.g., the LOGP [62] or DMM [141] models, with the d and $\mathbf{x}$ parameters.) We evaluate this extended model on the CRAY C90 and CRAY J90 vector multiprocessors, and we explore experimentally how various PRAM variants can be emulated in this model, with particular emphasis on the Queued-Read Queued-Write PRAM. The bulk of the thesis describes a set of case studies, which are used for exploring modeling and implementation techniques. The algorithms studied are sorting, merging, sparse matrix multiplication, and connected components. Each case study describes the selection of a practical PRAM algorithm, the process of mapping the algorithm onto the machine model, a performance model for predicting running time (which accounts for latency and contention), and practical issues and tradeoffs in developing fast implementations. In particular, we describe techniques for efficiently hiding latency on the CRAY C90 and CRAY J90 using vectorization. The experimental results in the thesis show that many irregular PRAM algorithms can be systematically implemented on high-bandwidth multiprocessors, resulting in fast and predictable performance.

In the remainder of this introduction, we motivate and describe our machine model, outline how PRAM algorithms are mapped to this model, describe our algorithm case studies, summarize the thesis contributions, and describe related work.

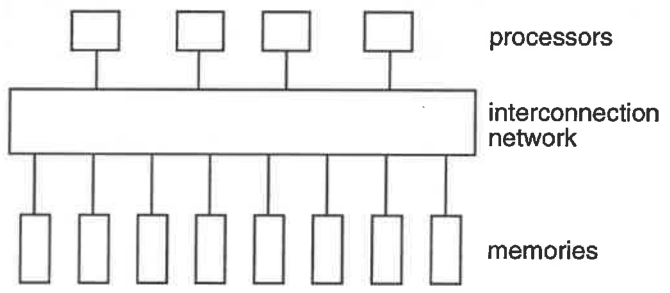


Figure 1.1: Machine model studied in the thesis.

1.1 Machine model: the Deluxe BSP

In recent years several models have been designed with the goal of hiding many details of parallel machines while still providing guidance in developing efficient algorithms. Examples of such models include the Bulk Synchronous Parallel (BSP) [201] and LOGP [62] models, which both aim to serve as high-level performance models of message-passing machines. These models provide a simple abstraction of the machine's interconnection network that ignores many details of the network, such as topology, while still capturing important aspects such as bandwidth and latency. The models help in optimizing algorithms, aid in understanding how algorithms scale, and guide choices among algorithms. They are often used in conjunction with experimentation to account for aspects that are not considered by the models, such as local computation times. As such, they have been quite successful, leading to practical designs of many algorithms [12, 21, 49, 62, 63, 83, 89, 98, 144].

We introduce and evaluate a model with similar goals, but for high-bandwidth shared-memory machines. Such machines include both vector multiprocessors, such as the CRAY C90 and J90, and multithreaded machines, such as the TERA MTA [6]. This work was motivated by the study of algorithms with irregular memory access patterns. In our analysis we found previous models either quite detailed, or inadequate for describing the key performance characteristics of the algorithms. For example, we found that a straightforward shared-memory variant of the BSP does not properly account for contention at the banks since there is no way to account for the relative speed of memory banks and processors. On the other hand, previous models of multibank memory systems [13, 17, 38, 41, 43, 47, 52, 66, 96, 152, 184, 185, 186, 199] are highly detailed, and the studies have considered only regular or random access patterns. We are interested in modeling algorithms with irregular, but not necessarily random, access patterns without requiring a complicated model. We are particularly concerned with capturing the effect of memory contention in these algorithms. We assume that the parts of applications with regular memory access patterns can be analyzed with more traditional approaches [52, 96].

The model we consider, a shared-memory variant of the BSP, assumes that a set of processors are connected through a pipelined interconnection network to a set of memory banks. (See Figure 1.1.) In addition to the three parameters of the BSP—the number of processors p, a *throughput* (bandwidth) parameter g, and the *periodicity* (latency and synchronization) parameter L—our model has two additional parameters: the *bank delay* d, and the *expansion factor* x. We refer to this model as the (d, x)-BSP (the "$\underline{\text{d}}$elu$\underline{\text{x}}$e" BSP). The bank delay models the throughput at a memory bank, and the expansion factor is the ratio of the number of memory banks to the number of processors. Table 1.1 lists several machines along with their expansion factors. Our experiments show that these additional parameters are necessary for taking account of high contention on the CRAY C90 and CRAY J90, machines for which the best sustainable gap, g, between memory requests issued by each individual processor is less than the best sustainable delay, d, between accesses to each individual memory bank. Chapter 2 shows experimentally that the (d, x)-BSP is a fairly accurate performance model for irregular access patterns on the CRAY C90 and J90, even though the model ignores details of the machines.

Machine	Procs (p)	Banks (B)	Expansion (x)
NEC SX-3	4	1024	256
Tera MTA	256	512×64	128
Cray C90	16	1024	64
Cray J90	16	256×2	32
NEC SX-4	32	1024	32
Convex C4	4	128	32
Meiko CS-2 node	2	16	8

Table 1.1: Expansion in the number of memory banks for various machines. In the current TERA design, memory banks are organized into memory modules containing 64 memory banks. The CRAY J90 memory banks are organized in pairs that share address and data paths. The Meiko CS-2 node contains two Fujitsu μ-VP vector processors.

1.2 Mapping PRAM algorithms to the Deluxe BSP

As we show in Chapter 2, the (d, x)-BSP is useful for modeling a class of high-bandwidth multiprocessors. It can also be used as a programming model. That is, an algorithm designer can explicitly determine how computation and memory accesses are mapped to processors and memory modules. However the primary goal of introducing the (d, x)-BSP was to understand the relationship of these machines to PRAMs, both in theory and in practice [31].

Instead of targeting the (d, x)-BSP directly, and thus incorporating latency and memory bank effects into algorithm design, we would like to use a higher level model that abstracts away some of the details, but still leaves us with the ability to reason about the performance on the (d, x)-BSP. In particular, we would like to find a systematic way of mapping a PRAM model onto the (d, x)-BSP.

Our approach to emulating PRAM algorithms is adopted from the BSP [201]. Latency is hidden by taking advantage of additional parallelism, or *parallel slackness.* Each physical processor simulates multiple *virtual processors*, which allows the global memory requests made by each physical processor to be pipelined. The simulation of virtual processors can be implemented with a number of hardware latency-hiding mechanisms including vectorization (as on the CRAY C90 and J90), multithreading, and prefetching.

In emulating memory access in a PRAM algorithm, there are two potential cause of contention at a memory bank: *module map contention*, that is, contention caused by the assignment of multiple PRAM memory locations to the same memory bank, and *memory location contention*, that is, simultaneous accesses to a particular memory location in the PRAM algorithm.

Module map contention can be addressed by using random mappings of locations to memory banks. (In special cases, we can use a regular distribution of locations to banks that minimizes contention, when such a distribution is known.) Chapter 2 quantifies the effect of random mappings on memory bank contention and shows that fairly accurate performance predictions can be made without considering module map contention, given sufficient memory bank expansion.

Memory location contention is addressed by selecting algorithms designed for an appropriate variant of the PRAM. Various PRAM models differ in their rules for simultaneous memory access; the Concurrent-Read Concurrent-Write (CRCW) model allows an arbitrary number of simultaneous accesses to any one memory location without penalty, whereas the Exclusive-Read Exclusive-Write (EREW) allows only one access at a time to a memory location. The Queued-Read Queued-Write (QRQW) PRAM allows for concurrent reading and writing to shared memory locations, but at cost proportional to the number of readers/writers to any one memory location.

In the thesis we consider algorithms designed for several variants, but the QRQW model is ideal for targeting the $(d, \mathbf{x})$-BSP in several respects. First, the QRQW accurately reflects the contention properties of most commercial machines [86]. In particular, the effect of bank delay in the $(d, \mathbf{x})$-BSP can be captured in the QRQW model by making the cost of simultaneous access proportional to the memory bank delay. Second, the QRQW PRAM is less restrictive than the EREW PRAM. From a theoretical standpoint, the QRQW is more powerful than EREW [86] and from a practical standpoint, some QRQW algorithms have lower constant factors than the corresponding EREW algorithms. And third, the QRQW PRAM can be efficiently emulated on the $(d, \mathbf{x})$-BSP, both in theory [31] and in practice.

Chapter 3 discusses a number of practical techniques for implementing PRAM algorithms, including techniques for implementing processor virtualization for various machine architectures, and for reducing memory contention and module map contention.

1.3 Algorithm case studies

The bulk of the thesis consists of a set of algorithm case studies. This section describes a common set of issues addressed in each case study and justifies the choices of algorithms studied. There are five key elements explored in each of algorithm experiment:

algorithm selection: We compare different algorithms for each of the problems and determine which algorithms are likely to be fast in practice, considering both asymptotic efficiency and constant factors.

mapping: We describe how the costs of latency and contention can be minimized when mapping the algorithm onto a pipelined-memory multiprocessor.

modeling: We derive performance equations in terms of machine and algorithm parameters, and analyze tradeoffs between latency, contention, and work.

implementation: We develop aggressive implementations, describe the key implementation issues, measure constants in each kernel, and compare measured performance with our predictions.

comparison: We compare our implementations to previous implementations, primarily in terms of generality and performance.

The irregular algorithms studied in this thesis include merging, sorting, sparse matrix multiplication, and graph connectivity. These algorithms, each of which is described in a separate chapter, are intended to serve as representative of the most basic irregular problems. Such core problems arise in a number of irregular applications and are often the bottlenecks in such applications. In addition, timings of optimized implementations exist for many machines, which allows us to draw informed quantitative and qualitative conclusions from the performance results. The specific algorithms studied in the thesis were selected for various reasons:

merging: Merging is a fundamental algorithm used in several applications, including merge sort, database join, set intersection, and divide-and-conquer computational geometry algorithms. Merging is an interesting case study because it is simple to describe, yet exposes several key issues. Issues explored include the choice between variants designed for various PRAM models, and tradeoffs between work-efficient algorithms and simpler, less efficient algorithms that have been previously used on vector and distributed memory machines.

sorting: Sorting has many applications in serial computing, and is arguably even more important in parallel computing. Sorting is a key substep in many parallel algorithms, including scientific applications such as particle simulation [64], as well as a variety of database and combinatorial algorithms. In addition, sorting can be used for simulating a variety of collective operations, including concurrent memory access, and sorting primitives are supplied in several parallel languages, including High Performance Fortran (HPF) [101] and NESL [24].

sparse matrix multiplication: The solution of sparse linear systems is an essential computational component in scientific computing. In particular, the solution of unstructured sparse linear systems is important in computational fluid dynamics, circuit and device simulation, structural analysis, and many other application areas. The most important kernel in many iterative solvers is sparse-matrix multiplication. In the case of unstructured problems, this kernel is difficult to optimize and typically dominates the computing time.

In addition to linear system solvers, other applications, including molecular dynamics and linear programming, can be formulated as sparse matrix multiplication [21].

connected components: Graph algorithms are fundamental to a variety of scientific and combinatorial applications. We focus on connected component labeling because it is a fundamental graph algorithm used as a substep in a variety of applications. For example, the graph connectivity problem is the dominant cost in simulating Ising Spin models using the Swendsen Wang algorithm [191]. In addition, graph connectivity is representative of a broad class of graph algorithms. Efficient PRAM parallel algorithms for connected components use many common primitive operations on graphs, such as neighbor operations, pointer shortcutting, nodes and edge deletion, and graph contraction and expansion.

The set of case studies covers a range of PRAM models: EREW for radix sorting, CREW and EREW for merging and for sparse matrix multiplication, and CRCW for connected component labeling. For algorithms that use concurrent memory access, we also consider variants for a queued access model.

The experimental work in this thesis was performed on the CRAY J90 and CRAY C90 [151]. We chose these machines because they provide a platform for exploring the benefits of having latency-hiding in combination with a high-bandwidth memory system with multiple memory banks. Perhaps the most important reason for choosing the CRAY vector machines is that they provide a platform for studying architectural characteristics that are likely to appear in future parallel architectures. In particular, future parallel machines are likely to supply much higher bandwidth communication, and recent commodity processor designs are starting to provide better support for hiding latency, via mechanisms such as prefetching. In addition, vector microprocessor architectures are an area of active research [131, 9, 205]. Although vector supercomputers are becoming less common, the results in this thesis are not strongly tied to the particular mechanisms used for hiding latency and could be applied to other machines that match our assumptions, such as the TERA MTA [6], a scalable, multithreaded shared-memory multiprocessor.

1.4 Contributions

This thesis makes several contributions:

- A *systematic* approach to algorithm design and implementation for high bandwidth multiprocessors, based on the QRQW PRAM. The thesis describes techniques for programming vector multiprocessors as PRAMs, using the concept of virtual processors for vectorization. The techniques apply to a broad class of regular and irregular PRAM algorithms.

Problem	Problem Size	Performance
sorting an array of keys	32M 64-bit keys	110 M keys/sec
merging two sorted arrays	16M 64-bit output keys	500 M keys/sec
sparse matrix-vector multiplication	9M nonzero elements	3800 M flops/sec
labeling connected components of a graph	40M nodes	115 M nodes/sec

Table 1.2: Performance summary of thesis case studies, indicating approximate throughput on a 16-processor CRAY C90. Detailed timings are given in Chapters 4–7.

- Fast, predictable, general-purpose implementations of sorting, merging, sparse matrix multiplication, and connected components PRAM algorithms for the CRAY C90 and CRAY J90. Table 1.2 presents a brief summary of timings for the algorithm implementations, several of which are fastest than all previous implementations.

- A model for high bandwidth pipelined-memory multiprocessors, that augments the BSP model with delay and expansion parameters. The model is detailed enough to make accurate performance predictions, yet abstract enough to include a broad class of current and future architectures using a variety of memory technologies and latency-hiding mechanisms. The thesis shows experimentally that that accounting for memory bank delay is necessary when contention is high, that the (d, x)-BSP is accurate on the CRAY J90 and CRAY C90, and that the QRQW PRAM is a good approximation to (d, x)-BSP.

1.5 Related work on implementing irregular algorithms

Our work differs from previous research in two basic areas: the model of parallel computation, as discussed in Section 1.1, and the approach toward implementing irregular algorithms. This section presents a high-level comparison to related work in implementing irregular parallel computation. Detailed discussion of related work on machines models is described in Chapter 2, and discussion of work related to the specific case studies (sorting, merging, sparse matrix multiplication, and connected components) is deferred to the relevant chapters.

1.5.1 BSP and LOGP implementations

Several irregular algorithms have been implemented using the BSP model, in areas such as computational geometry [65, 82], discrete-event simulation [44], plasma simulation [149], and sparse matrix computation [20]. In comparing our work to implementations of BSP algorithms, it is important to distinguish between the BSP model, and implementations based on BSP libraries [144, 89, 181]. The BSP model uses the gap parameter (g) parameter to quantify bandwidth limitations. In theoretical work describing emulations of PRAM algorithms on the BSP, sometimes the gap is assumed to have the ideal value, $g = O(1)$ [201]. In practical research using the BSP and LOGP models, the implementations and libraries target portability to a wide variety of machines [89, 61], from tightly-coupled MPPs with a moderate gap, to networks of workstations [149], with an extremely high gap. The limited bandwidth of these machines motivates practical work on design of new algorithms in which the primary focus is increasing locality [82, 65, 139, 63]. In contrast, by sacrificing portability and focusing on high-bandwidth machines, we are able to explore issues closer to one of the primary motivating factors for BSP, that is, practical implementation of PRAM algorithms.

1.5.2 Vector implementations

High bandwidth vector machines have been used for implementing a wide variety of irregular algorithms. Implementations on a single vector processor range from fundamental algorithms and data structures for garbage collection [8], dictionary operations [113], hashing [115], and matching primitives [177], to applications in areas such as symbolic circuit simulation [18], logic verification [104], computational geometry [189], n-body simulation [140], and particle simulation [137, 42]. A few parallel irregular algorithms have been parallelized for vector computers, including implementations of a sparse direct solver [206], a tree search algorithm for playing chess [111], and a randomized list-ranking algorithm [164].

There are three key differences between our implementation approach and most previous vector implementation studies. First, several implementations use an *ad hoc* approach to vectorization that is based on restricting the class of inputs or modifying inherently serial algorithms. (Two notable exceptions are the implementations of matching primitives [177] and list-ranking [164], both of which are based on simulating PRAM algorithms.) Second, most of the implementations use only a single vector processor. And third, nearly all ignore memory contention caused by irregular access patterns. In short, this thesis focuses more on systematic implementation and on scalability—not only in using multiple processors, but also in tolerating memory latency and avoiding memory bank contention, both of which become increasingly important as the number of processors is increased.

1.6 Thesis organization

The remainder of the thesis is organized as follows. Chapter 2 defines the $(d, \mathbf{x})$-BSP and describes a number of experiments that evaluate the accuracy of the model. Chapter 3 describes techniques for hiding latency and reducing contention, and also describes several basic algorithms and primitives. In Chapters 4–7, we describe a set algorithm experiments. Chapter 4 describes a mapping of an EREW PRAM radix sort onto pipelined-memory multiprocessors and highlights two issues: how to hide latency by simulating virtual processors and how to minimize module map contention by carefully distributing locations among banks. Chapter 5 compares several parallel merging algorithms and illustrates tradeoffs between work and latency. In addition, it quantifies the effect of concurrent reads in merging algorithms and demonstrates the importance of using a work-efficient algorithm. Chapter 6 describes a sparse matrix multiplication algorithm that fully hides latency for arbitrary sparse matrices. Chapter 7 describes algorithms for finding the connected components of a graph. The chapter emphasizes the importance of accounting for memory contention and demonstrates that a general-purpose PRAM algorithm performs well on regular and irregular graphs. Chapter 8 presents conclusions of the thesis.

Chapter 2

Modeling Pipelined-Memory Multiprocessors

The goal of this chapter is to provide a simple model for the design and analysis of irregular parallel algorithms on shared-memory multiprocessors with more memory banks than processors. For this purpose we extend Valiant's bulk-synchronous parallel (BSP) model with two parameters: a parameter for memory bank *delay*, the minimum time for servicing requests at a bank, and a parameter for memory bank *expansion*, the ratio of the number of banks to the number of processors. We show experimentally that the (d, x)-BSP captures the impact of bank contention and delay on the CRAY C90 and J90 for irregular access patterns, without modeling machine-specific details of these machines.

This chapter represents joint work with Guy Blelloch, Phil Gibbons, and Yossi Matias [31]. The chapter focuses on the definition of the model and experimental results. Theoretical results on PRAM emulations are presented in the published papers [31], and are summarized in Section 2.5.1.

2.1 Overview

In this chapter, we first show experimentally that the (d, x)-BSP model accurately characterizes the performance of the CRAY vector multiprocessors on irregular access patterns, even though the model ignores details of the machines.[1] We show this both for the CRAY C90, which uses static RAM (SRAM) and has a bank delay of 6 clock cycles, and for the more modestly priced CRAY J90, which uses dynamic RAM (DRAM) and has a bank delay of 14 clock cycles. The (d, x)-

[1]We assume, however, that the code is mostly or fully vectorized so that memory traffic is high. We do not claim that the model applies to programs that have large scalar components.

BSP has made it easier to analyze several algorithms and has allowed us to predict various effects that cannot be predicted without taking into account the bank delay. Although the (d, x)-BSP does not model the time exactly, due mostly to ignoring effects of the network, it more accurately captures the effect of high contention. Furthermore, the discrepancy between the BSP and (d, x)-BSP becomes larger as either the bank delay or the number of processors increases.

Second, we study to what extent the effects of multiple memory locations residing in a single bank can be ignored, when using random mappings of memory locations to memory banks. Many researchers have studied the effect of randomly mapping memory to banks (e.g., [6, 97, 116, 132, 141, 161, 162, 163, 201]). If there is sufficient parallel "slackness" (extra parallelism) so that each bank is receiving multiple requests, it has been shown [116, 141, 162, 201] that with high probability the memory references will be reasonably balanced across the banks. These results, however, assume that there is no contention at individual memory locations.[2] If we allow for contention at memory locations, then ignoring the mapping of memory locations to banks can significantly underpredict the running time even for unbounded slackness. This is because even when the number of memory requests is large, the number of *accessed* locations could be small and not well balanced across the memory banks. We study the impact of the expansion factor on the balance of requests, when only a small number of memory locations are accessed, and show that increasing the expansion factor reduces the effect of ignoring the mapping of memory locations to banks. For the CRAY C90, which has a high expansion factor, we show both experimentally and analytically that ignoring the mapping will underpredict running time by a factor of approximately 2.0 for a worst-case reference pattern and typically by much less. The effect of expansion for the CRAY C90 and CRAY J90 is shown in Figure 2.1.

Finally, we summarize results that explore scenarios under which two high-level models for algorithm design, the EREW PRAM (e.g., [112]) and the stronger QRQW PRAM [86] can be effectively mapped onto high-bandwidth machines (small g) when properly accounting for memory bank delay [31].

2.2 Accounting for memory bank contention and delay

Our model is an extension of Valiant's BSP model [201]. The BSP model was introduced to be a "bridging" model between software and hardware in parallel machines: software would be designed for this model and parallel machines would implement it. Such a standardized interface would allow software to be more easily ported to various hardware platforms. The BSP model

[2]In the case of Ranade's work [162] it is assumed that references to a single location are combined in the network.

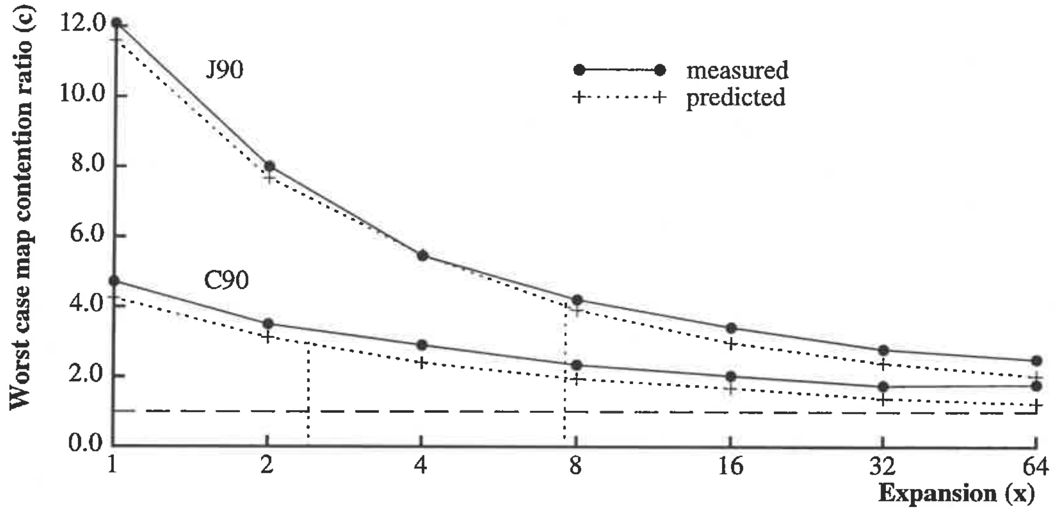


Figure 2.1: The ratio of the time that includes the effect of multiple memory locations being mapped to the same bank to the time that excludes the effect, when using random mapping. This is given as a function of expansion and is for a worst-case reference pattern. For both the CRAY C90 and J90 the actual expansion is 64 and the other data points are taken by using a subset of the banks. This shows the advantage of having an expansion greater than the d/g (shown with the dotted vertical lines) that is needed to match the servicing bandwidth at the banks with the issuing bandwidth at the processors.

consists of p processor/memory components communicating by sending point-to-point messages. The interconnection network supporting this communication is characterized by a throughput parameter g and a periodicity parameter L, where L is the sum of the global memory latency ℓ, and the cost of synchronization. The particular topology of the network is ignored and the cost to communicate among processors is assumed to be uniform, independent of the identity of the processors. A BSP computation consists of a sequence of "supersteps" separated by bulk synchronizations (typically, a barrier synchronization among all the processors). In each superstep the processors can perform local computations and send and receive a set of messages. Messages are sent in a pipelined fashion (i.e., each processor may issue messages and continue with its computation prior to the receipt of those messages). Messages sent in one superstep will arrive prior to the start of the next superstep. The time charged for a superstep is calculated as follows. Let W_i be the amount of local work performed by processor i in a given superstep. Let S_i (R_i) be the number of messages sent (received) by processor i. Let $W = \max_{i=1}^{p} W_i$, $S = \max_{i=1}^{p} S_i$, and $R = \max_{i=1}^{p} R_i$. Then the cost, T, of a superstep is defined to be

$$T = \max(W,\ g \cdot S,\ g \cdot R,\ L)\ .$$

Intuitively the communication throughput parameter, g, is the best sustainable gap between message sends issued by each individual processor; therefore $1/g$ represents the available bandwidth per processor. Intuitively the communication latency parameter, L, called the "periodicity factor" in [201], is the worst case time to deliver a message between two processors in an otherwise unloaded network plus the time to perform a barrier synchronization.

2.2.1 The (d, x)-BSP model

The (d, x)-BSP differs from the BSP described above as follows. The (d, x)-BSP is an explicit shared-memory model. Memory components (banks) are considered as separate from the processors, and their number is accounted for in the model. Instead of sending and receiving messages, processors make global memory requests that are serviced directly by the memory banks. To account for differences in the speed of memory requests by processors and responses by memory banks the (d, x)-BSP also assigns a distinct throughput parameter for the memory banks.

The (d, x)-BSP is depicted in Figure 2.2. It consists of p processors communicating by reading and writing memory words from a separate set of B memory banks; these memory banks are used as a shared memory by the processors. In practice these memory banks might be physically located next to the processors, but it is assumed that the processors are not involved in handling incoming memory requests. Processors also have local memory for use with local operations; this accounts for registers, cache memory, and main memory in each processor's local environment.

A (d, x)-BSP computation consists of a sequence of supersteps separated by barrier synchronizations. In each superstep the processors can perform local computations and make a set of pipelined, global memory requests. Requests made in one superstep will complete prior to the start of the next superstep. We include the same parameters as included in the BSP model but add two more: the *memory delay* and the *bank expansion factor*. The parameters of the (d, x)-BSP are summarized as follows:

p	number of processors
g	communication throughput parameter (gap)
L	the periodicity parameter (latency + synchronization)
d	memory bank throughput parameter (delay)
x	memory bank expansion factor (assumed to be ≥ 1)

where the number of memory banks, denoted as B, is $\mathrm{x} \cdot p$. We refer to a machine as (d, x)-*balanced* if $\mathrm{x} = d/g$. This is the point where the total bandwidth available at the processors and network for random access patterns matches the total bandwidth available at the memories. Intuitively, the gap parameter g is the best sustainable gap between memory requests (either reads or writes) issued by each individual processor. Intuitively, the periodicity parameter, L,

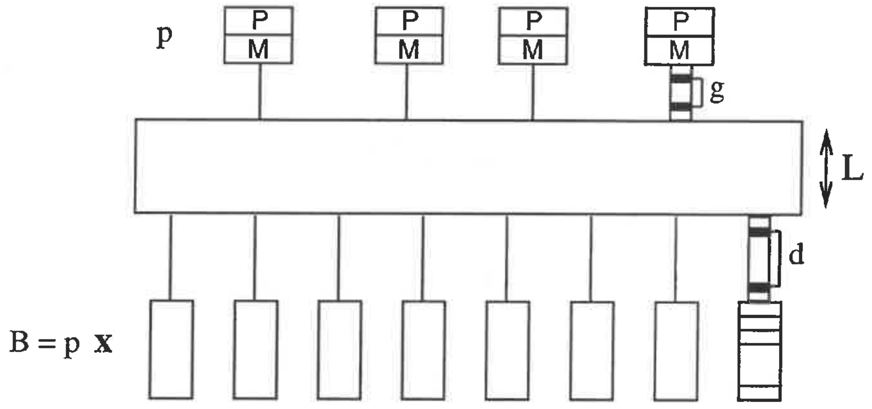


Figure 2.2: The (d, x)-BSP model.

is the worst case time to complete a single memory read in an otherwise unloaded memory system plus the time to perform a barrier synchronization. Intuitively, the delay parameter d is the best sustainable gap between memory requests serviced by an individual memory bank.

The time charged for a superstep is calculated as follows. Similar to the BSP, let W be the maximum amount of local work performed by any one processor in the superstep, and let S be the maximum number of global memory requests made (the maximum memory request load) by any one processor. Let R_j be the number of requests handled by memory bank j, and let $R = \max_{j=1}^{\mathrm{x}\cdot p} R_j$. Then the cost, T, of a (d, x)-BSP superstep is defined to be

$$T = \max(W,\ g \cdot S,\ d \cdot R,\ L)\ .$$

In our algorithm experiments, sometimes it is more appropriate to analyze algorithms in terms of the memory latency, without considering synchronization costs. In these cases, the parameter ℓ is used for the memory latency alone. Also, sometimes it is more appropriate to sum the costs of computation and memory contention, instead of using the maximum of these two costs. Summing can be more convenient for analyzing algorithms, and in some cases, for increasing the accuracy of performance predictions.

The contention at a memory bank can be due to not only the contention at a particular location in the bank, but also due to accesses to multiple locations within the bank. In the basic model we make no assumptions about how the memory locations are mapped onto the memory banks. This allows us to consider scenarios where the mapping is under user control and scenarios where the mapping is random. To separate the two types of contention we make the following definitions, each of which is defined relative to a single superstep. Let the *memory request contention*, k_i, to a location i be the number of requests to i, and let $k = \max_i k_i$. Let M_j be the set of locations mapped to memory bank j. The size of this set is the *module map*

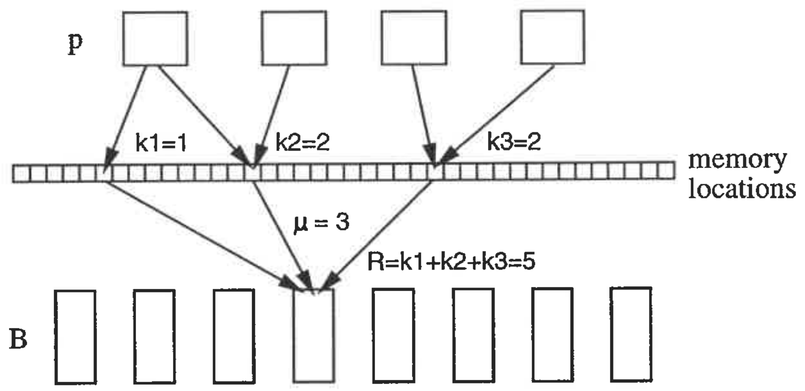


Figure 2.3: Memory request contention ($k1, k2, k3$), module map contention (μ), and module load contention (R).

contention, $\mu_j = |M_j|$, of bank j, and let $\mu = \max_j \mu_j$. Then

$$R_j = \sum_{i \in M_j} k_i$$

is the *module load contention* of bank j (see Figure 2.3).

2.2.2 Applicability of the model

The model assumes several properties of a machine. We now discuss the scope and limitations of the model.

First, the model assumes that each processor has enough outstanding memory requests to compensate for both network latency and bank delays. Allowing for multiple outstanding requests is relatively easy for memory writes, but more difficult for memory reads. Techniques for allowing multiple outstanding reads, often called latency hiding techniques, include vectorization, multithreading, prefetching, non-blocking caches, and other methods for decoupling the request for the memory from its use. Vectorization has been used for over 20 years to hide memory latency and has the advantage that it is simple to implement. On the other hand it restricts the kinds of programs that can be used. Multithreading was suggested and implemented for hiding latency on the HEP [182] and was later used in the design of the TERA MTA [6] and Sparcle [2]. Multithreading is more complicated to implement than vectorization but permits the use of a wider class of programs. Prefetching and non-blocking caches are becoming common on commodity processors, although the number of outstanding requests currently allowed (typically between 2 and 8 words or cache lines) probably cannot compensate for the latency to a large shared memory. If processors evolve so that they permit additional outstanding read requests, then it is quite possible that the model will apply to commodity processors attached to fast multi-bank memory systems.

Second, although the model accounts for contention at the processors and memory banks, it does not account for contention within the network. The BSP and LOGP models make similar assumptions. The definition of the g parameter accounts for network bandwidth under normal conditions, but for many networks, it is possible to construct particularly bad permutations.[3] In fact, in the next two sections we show that for the CRAY C90 and J90, the model breaks down under certain contrived conditions; however, under a random mapping of memory locations to banks, these conditions are very unlikely to occur.

Third, the model assumes that the memory banks are slower than the rate at which processors can issue memory requests into the network (i.e., $d > g$). With today's memory technology and processor speeds, the d parameter is typically on the order of 10 clock cycles. For the g parameter to be less than that, the bandwidth per processor into the network needs to be quite high.

Fourth, we assume that in each superstep of the (d, x)-BSP, each processor injects its memory requests into the network in a random order (although in practice we find this is not necessary). This assumption is made since even if the requests are reasonably distributed among the banks within the whole superstep, they might be badly distributed over time during the superstep. If we assume infinite buffering in the network and at the banks and assume that requests can overtake each other, then this may not be a problem, but most networks have only limited buffering. The limited buffering can cause congestion that will back up future requests even though they are going to a non-congested destination. Processor-memory feedback effects can compound the problem [184]. Our experiments on the CRAY J90 have shown slowdowns of over a factor of 20 using bad injection orders as compared to random injection. We note, however, that in practice it is often not necessary to spend extra time randomly ordering requests since it is known that the requests are well distributed.

Fifth, the (d, x)-BSP does not explicitly model the processors' local environments, including cache behavior and local arithmetic operations. As with the BSP and LOGP models, the (d, x)-BSP focuses on the interprocessor communication aspects of parallel machines, as these are presumed to be the primary bottlenecks of parallel programs. Since the (d, x)-BSP can model the local work only within a rough estimate, experiments are needed to get an accurate measure of this component. Typically a small experiment will suffice to get an accurate prediction of work over a range of problem sizes and number of processors [34, 62].

Another consideration regarding the local environment is in accounting for the use of caches. In cache-based symmetric multiprocessors (SMPs), understanding the cache behavior is often necessary in order to obtain reasonably accurate performance prediction. In the (d, x)-BSP it is

[3]As discussed in the next section, we derive the g parameter assuming a random permutation of addresses. For certain regular permutations, the time through the network could be better than predicted.

up to the user to determine, for each shared memory reference, whether the value is present in its local cache or must be retrieved from the memory (or some other processor's cache). A local cache hit is accounted as a local operation; a cache miss is accounted as a global operation. Different machines have different cache policies, and the accuracy of the $(d, \mathbf{x})$-BSP prediction depends in part on the extent to which operations can be properly accounted as local or global. Furthermore, it might be necessary to account for memory traffic caused by the cache coherence protocol itself. Considerations of cache behavior or other uses for the local memory provided by the $(d, \mathbf{x})$-BSP do not arise in our experiments on the CRAY C90 and J90, since these machines have adequate memory bandwidth, limited local memory, and no caches for vector data.

Finally, the model does not take account of the possibility of caching at the memory banks, as available in the design of the TERA MTA [6], and as suggested by Hsu and Smith [110]. Extending the model to account for caching at the bank is an interesting area of future work.

2.3 Case study: modeling the Cray C90 and J90

This section presents a qualitative and quantitative comparison of the CRAY C90 and CRAY J90 to the $(d, \mathbf{x})$-BSP. The experiments in this section provide evidence that the abstract model can produce realistic predictions of running times despite ignoring many architectural details, such as the use of vector processors, the topology of the interconnection network, the flow control for the network, and the priority scheme on the banks. The experiments also show some cases where the model breaks down. The features of the CRAY vector machines that make them suitable for the model are the following:

1. The use of vector gather and scatter instructions to allow for multiple outstanding memory requests to an arbitrary set of addresses.

2. A high bandwidth interconnection network between the processors and the memory banks that supports fine-grained memory references.

3. Memory banks that are significantly slower than the rate at which processors can issue memory requests. The ratio d/g is 5 on the C90 and 7.7 on the J90.

However, to use the $(d, \mathbf{x})$-BSP model for the CRAY vector machines we need to separate the costs of regular versus irregular accesses. In particular on the CRAY since the bandwidth for unit stride accesses is very high (the bandwidth for two loads and a store is the same as for an add) and the load across the banks is perfectly distributed for such accesses, we count such unit stride accesses as part of the work W.

Here we discuss the features of the CRAY C90 and J90 in more detail. The instruction set supports vector load and store instructions, which load or store up to 128 words per instruction

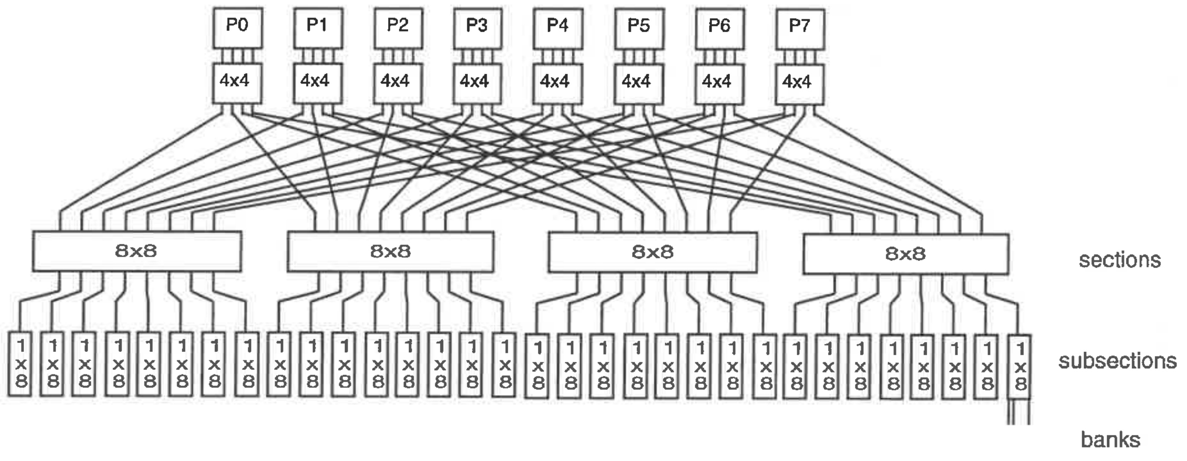


Figure 2.4: Cray Y-MP interconnection network (adapted from Smith and Taylor [184]).

(64 on the CRAY J90). These instructions can either be strided for regular access patterns, or can be based on a vector of addresses for indirect addressing. In the indirect loads and stores, often called *gather* and *scatter*, each address specifies the location of a single 64-bit word allowing for fine-grained memory accesses. The processors are connected to the memory banks with a multistage network (see Figure 2.4). In the largest memory configuration of the CRAY C90, banks are divided into 8 "sections" at the first level, which are each further subdivided into 8 "subsections", each of which contains 16 memory banks. Each processor has independent paths through the section and subsection, but processors can interfere with each other due to bank conflicts. Sequential memory locations are interleaved across memory banks, making regular, strided memory access fast (except on strides that are large powers of two).

The latency through the network is approximately 9 clock cycles each way. When added to the access time at the memory bank, the total latency for a load is between 23 and 35 clock cycles, depending on the CRAY model and assuming no contention at the memory bank. This latency is usually hidden when using vector loads and stores since each load or store requests multiple locations that are pipelined through the network. Furthermore, vector memory references can overlap so that one batch of 128 loads or stores can start before the previous has finished.

The CRAY has multiple memory ports per processor (2 on the J90 and 6 ports on the C90). These ports have different functions. Some are for reading and some are for writing, and only a subset of the ports can be used for gathers and scatters (1 on the J90 and 2 on the C90). This means that the bandwidth for irregular access patterns is not as high as for regular access patterns.

	C90	C90*	J90
Processors	16	16	16
Banks	1024	512	1024
Memory	SRAM	SRAM	DRAM
Clock period	4.2 nsec	4.2 nsec	10.0 nsec
g	1.2	2.5	1.8
d	6	6	14

Table 2.1: The parameters for the CRAY C90 and CRAY J90. The gap and delay are measured in clock periods. The C90* is the configuration of the C90 available to us at the Pittsburgh Supercomputing Center, which has only half the memory banks, memory ports and network of a full configuration.

Table 2.1 shows the (d, x)-BSP parameters for the CRAY C90 and CRAY J90. The gap g is measured experimentally and the other values are available from the machine specifications. We are interested in the gap for irregular access patterns, in particular ones that require gathers and scatters. As mentioned earlier the regular accesses can be counted in the work term.[4] We base the gap on the time for a scatter to random locations using all the processors, where the destination addresses are loaded from memory. The time for a gather is almost the same (within 10 percent). Our measured gap is somewhat lower than the theoretical peak performance for gather or scatter operations due to the fact that the memory system is fairly saturated. Bucher and Simmons have also noted this saturation effect [41]. Our experiments show that the gap measured for random access patterns reasonably model other irregular patterns.

We performed several experiments of memory access patterns to quantitatively compare the predictions given by the (d, x)-BSP with running times on the CRAY C90 and CRAY J90. The experiments were selected to test various aspects and extremes of the model. Figure 2.5 summarizes the experiments. For most experiments the measured times closely match the predicted times. For one of the experiments, 4c, the measurements differ by up to a factor of 2.5 due to the effects of the network. In all our synthetic experiments we assume that the work term W can be ignored since on the CRAY it is typically subsumed by the $g \cdot S$ term. In the algorithm experiments described in Chapters 3–7, the work term is measured experimentally.

For all experiments we randomize the injection of memory requests within the processors. All the experiments are based on using the scatter operation, although experiments with the gather operation have given almost identical results. The patterns we are interested in cannot be created with strided access—timings for various strided access patterns can be found

[4]In our experiments all regular loads and stores are unit stride.

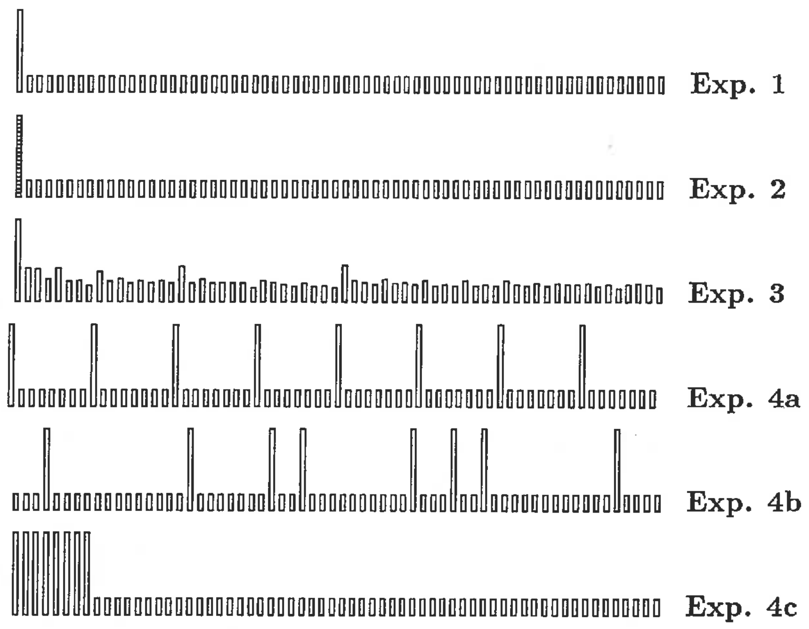


Figure 2.5: Summary of the experiments. Each bar represents the load on one memory bank. A shaded bar (leftmost bar in Exp. 2) represents multiple different locations being written to the bank while a clear bar (all others) represents a single location being written.

elsewhere [52, 186]. All experiments were run on a dedicated 8-processor system. All graphs are for the CRAY J90 except where noted—CRAY C90 results are qualitatively similar. For all experiments $S = 64K$ and the periodicity parameter L is negligible.

Experiment 1: The first of the experiments is used to verify the (d, x)-BSP time equation $T = \max(g \cdot S, d \cdot R)$ over a range of R. The experiment consisted of writing one location with load R; the remaining work is spread across $B - 1$ memory locations, one memory location per remaining bank. R is varied and S is kept constant. For this experiment, the model is accurate over a range of contentions, as shown in Figure 2.6. However, at the knee of the curve, the measurements for the CRAY J90 are slightly higher than predictions due to small additive effects of the memory and processor terms.

Experiment 2: The second experiment is used to verify that the time is determined by the maximum contention at a bank, independent of whether all the contention is to one location within the bank or is to many locations. The experiment consisted of sending a single request to R different locations within a single bank; the remaining work is spread across $B-1$ memory locations, one memory location per remaining bank. Again, R is varied and S is kept constant.

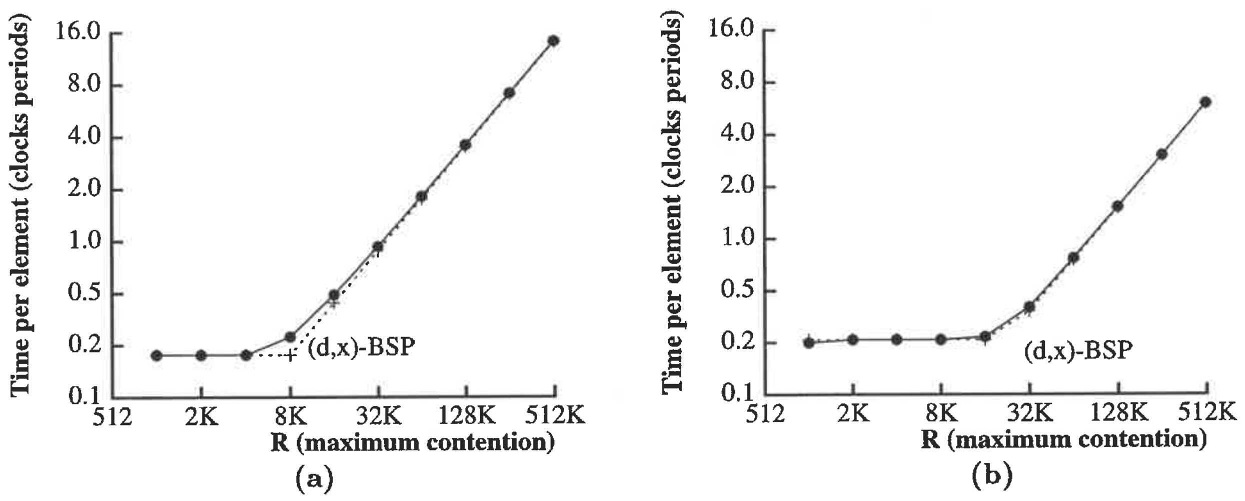


Figure 2.6: Experiment 1: Measured and predicted times on (a) the CRAY J90 and (b) the CRAY C90 over a range of contentions (log-log scale). The measured time (shown with a solid curve) is close to the maximum of the time spent at the processors and memory. The knee in the curve is where the dominant term switches.

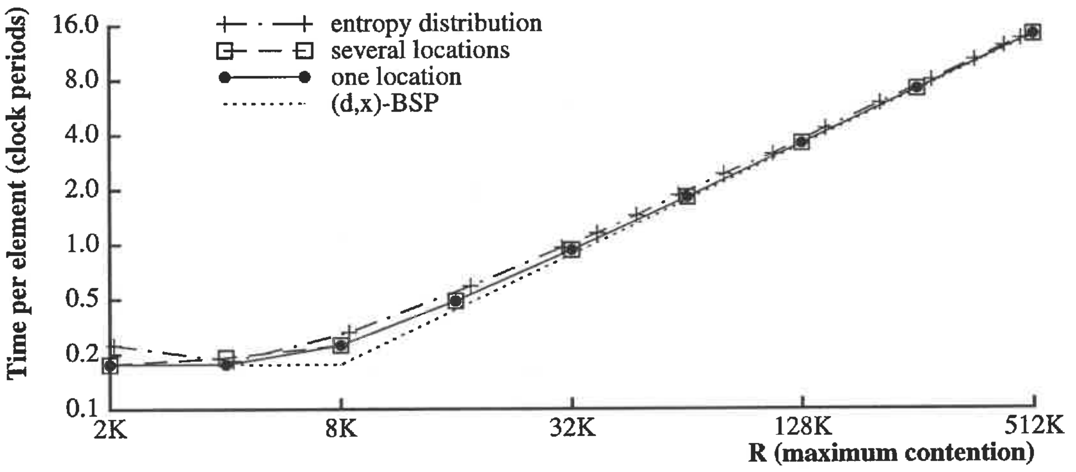


Figure 2.7: Comparison on the CRAY J90 of experiment 1, experiment 2, and experiment 3: one measuring the time where each bank contains at most one active memory location, one where the accesses to each bank are spread over many memory locations, and one using successive ANDings of random keys. As expected, the curves are nearly identical.

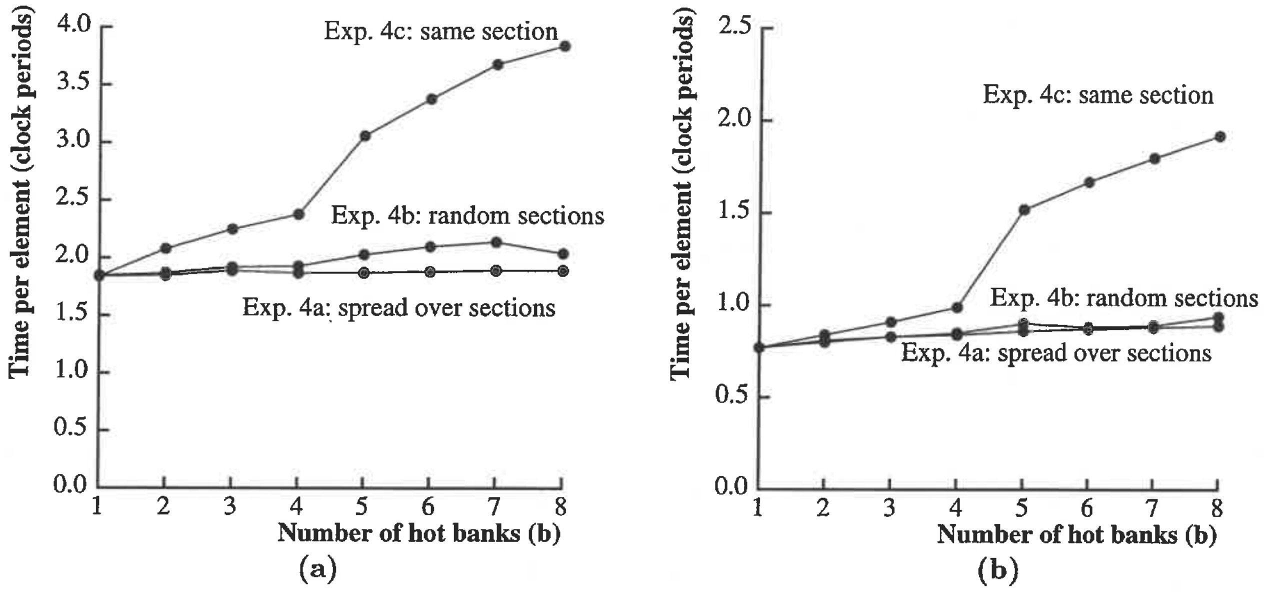


Figure 2.8: Time on (a) the CRAY J90 and (b) the CRAY C90 as a function of the number of hot banks when R is constant ($R = S \cdot p/8$) The three curves are for different distributions of hot banks across the network and show the effect of network contention.

This differs from the previous experiment only in that the load on the "hot" bank is due to multiple memory locations rather than multiple elements to the same location. The results shown in Figure 2.7 verify that the performance is not affected by whether the R term is dominated by location contention or module map contention.

Experiment 3: To verify that the running time can be accurately predicted for less regular distributions of memory accesses, we constructed an experiment using the entropy distributions suggested by Thearling and Smith [194]. The distributions are generated by starting with a set of random keys and then bitwise ANDing together each key with another key selected at random. Iterating this process generates a family of distributions each with a higher contention than the previous and eventually all keys become 0. The experiment was run over the whole family. Figure 2.7 shows the time as a function of this maximum contention. The predictions are slightly less accurate than those in Experiments 1 and 2, but are still within 30 percent of the measurements.

Experiment 4: To verify that the running time is determined by the maximum contention R, independent of the average contention and distribution of contention, we send R requests to one location within each of b banks and send an equal portion of the remaining work to $B - b$ locations all in different banks. R and S are kept constant, while b is varied. We tried three versions of this experiments differing in how the high-contention banks are distributed across the network: (a) evenly distributed across the network, (b) randomly distributed across the banks, and (c) all within the same section of the network. Figure 2.8 shows the results of the experiment using the worst-case value for R. Versions (a) and (b) are quite close to the predicted performance. Version (c), however, is up to a factor of 2.5 off from the prediction because of congestion at one of the *subsections* of the network. A more refined model would be needed to take account of this [184, 185], but the experiment shows that even in what we expect to be the worst case the predictions are not catastrophic. Note that when memory is mapped at random into banks, an issue that is discussed in the next section, the situation described in version (c) is unlikely.

2.4 Using random memory mappings

Randomly mapping memory locations to memory banks is a standard technique to reduce module map contention (contention due to multiple memory locations being mapped to the same bank) in emulations of shared memory on machines with a fixed set of memory modules (see, e.g., [6, 86, 97, 116, 132, 141, 161, 162, 163, 201]). The primary advantage of random mapping is that it ensures that concurrently requested memory locations will likely be distributed evenly across the banks. In this section we study to what extent we can ignore the module map contention (μ) when randomly mapping memory to banks. In particular we consider the ratio of the time including module map contention to the time excluding it. We call this ratio the *map contention ratio* (C), and in the (d, x)-BSP it can be expressed as

$$C \;=\; \frac{\max(W,\ g\cdot S,\ d\cdot R,\ L)}{\max(W,\ g\cdot S,\ d\cdot k,\ L)} \tag{2.1}$$

We are interested in how bad this ratio is for various machine parameters (p, g, d, x) and memory access patterns. We show that for the CRAY, which has a reasonably high expansion factor, C is small. The results in this section are generalized in the context of the QRQW PRAM emulation in the next section.

To derive an equation that bounds the map contention ratio we consider the memory access pattern in which all memory locations that are being accessed have the same contention. This pattern seems to maximize the map contention ratio for a given k (maximum contention at any memory location). We call this a *uniform distribution* of requests, and assuming each processor is making S requests, then a total of $m = Sp/k$ locations are being accessed. We are interested in the map contention ratio as a function of m—the worst ratio over m will give us the worst overall ratio. If we assume the cost is dominated by the send and contention terms we can simplify equation 2.1 to

$$C(m) = \frac{\max(mz, \mu)}{\max(mz, 1)}$$

where $z = g/dp$; z is a constant of the machine. The module map contention μ can be expressed in terms of the function $Urn(m, n)$, which is the expected maximum number of balls in an urn when throwing m balls into n urns at random. More specifically, let m balls be thrown independently at random into n urns, and let y be the random variable representing the maximum number of balls in any one urn; then, $Urn(m, n) = \mathbf{E}(y)$, the expected value of y. We can estimate:

$$C(m) \approx \frac{\max(mz, Urn(m, B))}{\max(mz, 1)} \tag{2.2}$$

Figure 2.9 shows both the predicted and measured values of C as a function of m for the CRAY C90 and the CRAY J90. The measured ratio is based on the average over 20 trials. The results show that for all m, $C(m)$ is at most 2.0 on the C90 and 2.5 on the J90, and for most patterns it is close to 1. This suggests that for many practical purposes we can ignore the module map contention when using random mappings on the CRAY—inaccuracies in predictions from other sources are likely to dominate. Intuitively, the reason for the peak in the graph is that as we increase the number of "hot" locations (m) past the peak, the load at each location decreases and eventually the first term in the equation $\max(g \cdot S,\ d \cdot k)$ dominates. As we decrease m below the peak, it becomes less likely that multiple locations will be mapped to the same bank. In particular when $m < \sqrt{B}$ it becomes less likely that more than one location will be mapped to a single bank. The slight difference between the predicted and measured times is due to effects in the network as discussed in the previous section. In particular, at these small m it is reasonably likely that the locations are not only imbalanced across the banks, but are also imbalanced across the *sections* in the network, causing backup at the source due to section conflicts.

It is more interesting to consider the effect of the machine parameters on the worst case map contention ratio. Equation 2.2 is maximized when $mz = 1$ (i.e., $m = dp/g$), giving

$$C_{\max} = Urn\left(\frac{1}{z}, B\right) = Urn\left(\frac{dp}{g}, p\mathrm{x}\right) .$$

Figure 2.1 shows $C_{\max}$ as a function of x for d and g set to the parameters of the CRAY C90 and CRAY J90 and p set to 16. As can be seen, it is helpful to have an expansion factor beyond the (d, x)-balanced ratio ($\mathrm{x} = d/g$) in order to minimize the impact on the running time of module map contention.

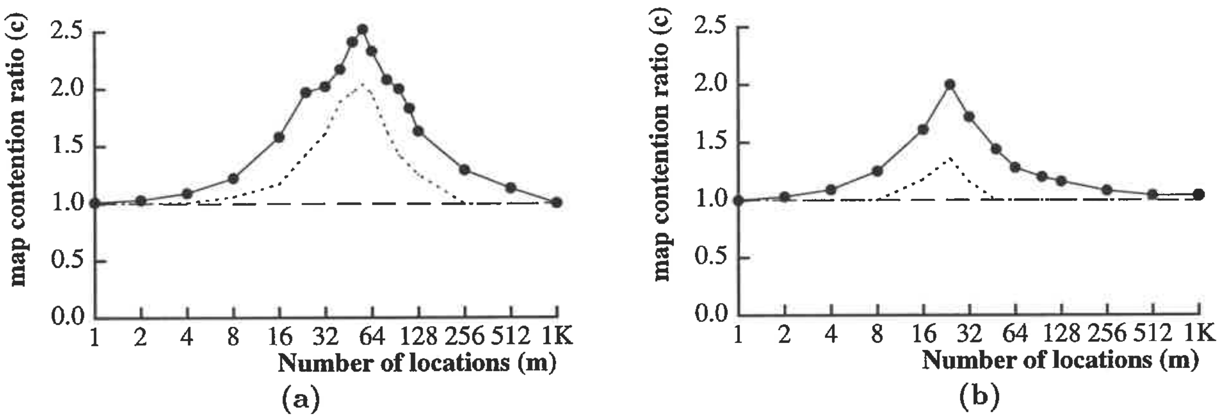


Figure 2.9: The measured and predicted map contention ratio C for (a) J90 and (b) C90. The measured ratio (solid curves) is taken as the ratio of the measured running time on the CRAY to the equation $\max(g \cdot S,\ d \cdot k)$, which ignores module map contention. The predicted ratio (dotted curves) is given by Equation 2.2. This is for a uniform distribution to m locations using random mappings from locations to banks. S is kept constant; $p = 8$.

2.5 High-level programming model: the QRQW PRAM

In this section we explore scenarios under which a high-level model for algorithm design, the QRQW PRAM, can be effectively mapped onto a (d, x)-BSP and hence onto high-bandwidth machines.

The QRQW PRAM [86] is a variant of the well-studied PRAM model (see, e.g., [112, 117]) that allows for concurrent reading and writing to shared memory locations, but assumes that multiple reads/writes to a location queue up and are serviced one at a time (named the "queue-read queue-write (QRQW)" contention rule in [86]). Specifically, the QRQW PRAM consists of p processors communicating by reading and writing words from a shared memory. Processors also have local memory for use with local operations. A QRQW PRAM computation consists of a series of supersteps separated by barrier synchronizations. In each superstep the processors can perform local computations and make a set of pipelined, global memory requests. Requests made in one superstep will complete prior to the start of the next superstep. The time charged for a superstep is calculated as follows. Let W be the maximum amount of local work performed by any one processor in the superstep, let S be the maximum number of global memory requests made by any one processor, and let k be the maximum number of requests to any one location. Then the time for the superstep is $\max(W, S, k)$.

The QRQW PRAM is an even simpler model than the (d, x)-BSP. Unlike the BSP or (d, x)-BSP models, the QRQW PRAM memory is not explicitly partitioned into memory banks—each processor has equal access to each memory location. Furthermore the QRQW PRAM has no

g or L parameters. The emulation of the QRQW PRAM on the (d, x)-BSP hides the latency and synchronization cost L by using a factor of at least L more "virtual processors" on the QRQW PRAM than are available on the (d, x)-BSP. The QRQW PRAM is more powerful than the well-studied EREW PRAM (which requires $k = 1$ at each step) but less powerful than the well-studied CRCW PRAM (which permits arbitrary k without charge). It was argued in [86] that the QRQW contention rule more accurately reflects the contention capabilities of most machines than the EREW or CRCW contention rules. An interesting question is under what conditions can one use the simpler QRQW PRAM instead of the (d, x)-BSP for modeling algorithms.

2.5.1 Overview of theoretical emulation results

Two goals motivated the development of the (d, x)-BSP. One goal, modeling irregular algorithms and implementations, is explored in the thesis. A second goal is to understand and formally prove the relationship between PRAM models, such as the QRQW and EREW PRAMs, and high bandwidth multiprocessors. Here, we give a brief overview of theoretical emulation results, in order to provide more context for our experimental study of PRAM algorithms.

Recall that a machine is (d, x)-balanced if $x = d/g$, that is, the total bandwidth available at the processors and network matches the total bandwidth available at the memory banks. Let d_g be the bank delay normalized to the gap parameter, i.e., $d_g = d/g$. In [31], we present two emulations of the QRQW PRAM on the (d, x)-BSP, one for the case where $x \geq d_g$ and one for the case where $x < d_g$. In the former case, we observe that any step-by-step emulation must incur an overhead of g in the work performed, and we provide an emulation of the QRQW PRAM on the (d, x)-BSP that matches this work overhead. Thus, when g is a small constant, as when modeling high-bandwidth machines, the emulation is work-optimal. The slowdown in the emulation is a nonlinear function of the parameters of the (d, x)-BSP; the slowdown is minimized when $x \geq d_g \cdot 2^{d_g}$, in which case d_g is only an additive term in the slowdown. As for the case when $x < d_g$, we observe that any emulation must also incur overhead due to the insufficient bandwidth at the memory banks, and we provide an emulation whose work bounds match the lower bound that we prove.

The emulation results (upper and lower bounds) apply as well to the EREW PRAM. These results extend the previous results in [201] and [86] that showed that when g is a small constant, there is a work-optimal emulation of the EREW PRAM and QRQW PRAM, respectively, on the original BSP model in which the slowdown in the emulation is $\Theta(\lg p + L)$. The new emulations, like these previous emulations, are randomized emulations, where the emulation is always correct and the upper bound on the slowdown in the emulation is achieved with high probability (w.h.p.), i.e., for any prespecified constant $c > 0$, the bound is achieved with probability at least $1 - p^{-c}$.

More recently, the high-level Queuing Shared Memory (QSM) model was shown to have a work-preserving emulation on the (d, x)-BSP as long as $\mathrm{x} \geq d/g$ (without restrictions on g) [88].

2.6 Discussion

This chapter studies the effectiveness of using the (d, x)-BSP as a simple model for shared-memory machines to analyze the memory performance of algorithms. The model accounts for memory bandwidth, memory latency, memory bank delay, and memory bank expansion. We verified that the (d, x)-BSP reasonably explains the memory performance of the CRAY C90 and J90 on irregular access patterns. We have focused on modeling the effect of these features on unstructured computations, where the memory access patterns are irregular and the lack of locality of memory reference stresses the bandwidth limitations of the shared memory machine. Although the (d, x)-BSP abstraction hides many machine-specific details, our results show that it still gives useful guidance to algorithm designers by providing performance predictions that are reasonably accurate.

Chapter 3

Techniques and Basic Algorithms

This chapter describes an approach to implementing PRAM algorithms on high-bandwidth pipelined-memory multiprocessors. It brings together useful techniques and basic algorithms from several contexts and adapts them to a common framework that is used in later chapters describing algorithm experiments. Section 3.1 describes implementation issues, in particular, techniques and tradeoffs for hiding latency on various categories of architectures. Section 3.2 describes our approach to developing practical low-contention algorithms and describes various techniques for reducing contention. Section 3.3 presents algorithms for binary search, scan primitives, and random permutation. These algorithms illustrate the techniques described in this chapter and serve as building blocks for designing more complicated algorithms. Finally, Section 3.4 highlights the application of various techniques discussed in this chapter to the algorithm case studies in the remainder of the thesis.

3.1 Implementing latency-hiding with virtual processors

In order to hide latency when implementing PRAM algorithms, we use a general approach based on the BSP model [201]. We use *parallel slackness*, that is, algorithms are implemented using more *virtual processors* than physical processors. Parallel slackness allows memory access to be scheduled and pipelined. Specifically, when simulating a parallel algorithm with v virtual processors per physical processor, there are v memory addresses available at the beginning of each parallel step. The memory operations can be issued in a pipelined fashion, thus amortizing the memory latency over v memory operations.

For some classes of algorithms and programs, latency can be hidden automatically with a combination of compiler optimizations (such as vectorization and compiler-directed prefetching) and transparent hardware mechanisms (such as nonblocking caches and write buffers). Automatic techniques are effective with many regular algorithms, but are less effective on irregular algorithms. The main advantage of hiding latency explicitly with virtual processors is that this technique can be applied to a wide variety of regular and irregular algorithms and can be implemented with a variety of latency-hiding mechanisms, including multithreading, prefetching, and vectorization.

There are many variations of pipelined-memory architectures. (See Lenoski and Weber [134] for a general overview of scalable shared-memory machines.) Our presentation classifies pipelined-memory architectures into three categories: multithreaded processors, pipelined microprocessors, and vector processors. In the following subsections, general issues in latency-hiding are described for each of these categories. (An example of mapping a histogram algorithm onto each type of machine is given later in Section 4.3.1.) To provide background for the experimental results on vector computers presented in later chapters, vector architectures will be discussed in more detail than other architectures.

3.1.1 Mapping to multithreaded processors

With *multithreading*, memory latency is hidden by switching context to another thread. Highly-parallel PRAM algorithms can be executed efficiently on a multithreaded machine by simulating multiple PRAM processors and assigning a thread to each virtual PRAM processor. There are, however, many variations in the implementation of multithreaded architectures [1, 37, 134], and these variations lead to subtle implementation tradeoffs. (Boothe and Ranade classify several variations of multithreading [37], and Agarwal considers various performance tradeoffs in multithread processors [1].) From the standpoint of the implementing PRAM algorithms, the key issues are:

context-switch granularity: Some machines, such as the TERA [6, 5] (which is based on HEP [182] and Horizon [128]), can switch context on every clock cycle, whereas some multithreaded architectures delay context switching until a memory access is issued, or until the result of a memory load is unavailable. The context-switch granularity will influence the amount of parallel slackness needed for efficient execution.

synchronization costs: The amount of slackness needed will also be influenced by synchronization costs. Some multithreaded machines provided hardware support for synchronization (e.g., with fetch-and-add [6]). The synchronization costs will also be affected by the mechanisms for thread scheduling. Depending on the system, the scheduling and mapping of threads to processors can be handled automatically by the compiler and run-time system [5], or scheduled explicitly by the programmer.

local memory: In contrast to custom multithreaded processors, some multithreaded processor implementations are based on augmenting microprocessor designs. Examples include

Sparcle [2] and simultaneous multithreading [198], a technique for permitting several independent threads to issue instructions to a superscalar processor. With microprocessor-based designs, several other issues are raised, as discussed in the next section.

3.1.2 Mapping to pipelined microprocessors

A few of the general implementation issues to be considered for microprocessors are:

locality: With microprocessor-based designs, caches can be used both to reduce global bandwidth requirements and to reduce memory latency. When simulating virtual PRAM processors on cache-based memory architectures, two key issues are the global memory bandwidth, and the characteristics of caches or local memories. While latency-hiding mechanisms such as multithreading and prefetching can hide memory latency, they cannot overcome global bandwidth limitations, and in fact, can increase the bandwidth requirements due to cache interference between virtual processors. If global bandwidth is limited (i.e., if the gap is high), then a direct implementation of a PRAM algorithm will be inefficient, unless the algorithm has a high degree of inherent locality. A second important issue is the memory latency to cache or local memory. In particular, if this latency is sufficiently small, it may not be necessary to use parallel slackness for embarrassingly parallel local computation.

memory granularity: The $(d, \mathbf{x})$-BSP model assumes fine-grained (e.g., single word) memory granularity, but most microprocessors use larger cache lines (typically 32 to 128 bytes). For algorithms that use fine-grained memory access, bandwidth might be wasted when only part of a cache line is used. More importantly, fine-grained random access can cause additional inter-processor communication for maintaining cache coherence if the size of items being written to memory is smaller than the unit of sharing.

large address spaces: Support for large address spaces is important for executing many PRAM algorithms, e.g., for algorithms with pointer-jumping. On some microprocessors, random accesses to virtual memory can cause significant TLB miss penalties. We assume that these can be avoided, for example, with hardware and operating system software support for large page sizes [80, 125].

latency-hiding mechanisms: There are a variety a mechanisms for latency hiding, including prefetching, nonblocking caches, write buffers, and pipelined uncached accesses. When non-blocking caches are combined with speculative execution, some degree of prefetching can be performed automatically. The particular latency-hiding mechanism will influence the amount of parallel slackness needed for efficient execution. Note that quantifying the benefits and costs of latency-hiding can be beneficial even for implementing PRAM algorithms on a single pipelined processor [203].

memory semantics: The implementation of parallel slackness is also influenced by the memory access semantics. With a non-binding memory request, the request has no effect on the semantics. For example, if the value in a memory location is modified between a prefetch instruction and a load instruction, the load instruction will retrieve the updated value. With a binding request, the value read is determined at request time; for example with a non-blocking cache, the value is determined at the time of the load instruction, rather than the time the value is used. Some architectures leave the binding undefined: the value read can be any value held in the memory between the initiation and completion of a read, for example, in some vector architectures and decoupled execute/access architectures [183]. Implementing algorithms by simulating virtual processors avoids many of the issues about memory semantics because, typically, there are no dependences between operations executed by the virtual processors in an algorithm step.

Of commercial microprocessor-based machines, the best match to the $(d, \mathbf{x})$-BSP is the Cray T3E [175], which efficiently supports pipelined global access to a large address space. Processors are augmented with external *E-registers*, a set of 512 memory-mapped registers for highly-pipelined, fine-grained access to global memory. Dozens of requests per processor can be simultaneously outstanding.

3.1.3 Mapping to vector processors

We begin by briefly describing the key features of a typical register-based vector multiprocessor architecture. (See [107, 99] for additional background material.) A typical vector processor contains a scalar processor augmented with several vector registers (each holding up to $V_{\max}$ elements, where $V_{\max} = 128$ on the CRAY C90) and vector instructions for operating on the vector registers (e.g., to add two vector registers). Vector instructions are implemented using highly-pipelined functional units. Typically, there are multiple independent functional units, and these units can be chained together in order to provide the output of one functional unit directly to the input of another. Memory instructions are provided for loading and storing strided vectors of up to $V_{\max}$ elements, and for indirectly loading and storing vectors. On some vector processors, the memory ports act as functional units and allow vector memory instructions to be chained with other vector instructions, including other vector memory instructions.

A few of the key issues to be considered for simulating PRAM algorithms on vector multiprocessors are:

latency tolerance: The pipelined implementation of the functional units and memory system imposes a latency for vector instructions. Functional unit latencies for vector arithmetic operations are typically in the range of 5–20 clock periods, and memory latencies for vector memory operations are typically 20–100 clock periods. These latencies can be hidden by issuing vector instructions with long vector lengths (as close to $V_{\max}$ as possible). Tolerating hardware latencies is becoming more important because recent vector multiprocessor designs (such as the CRAY J90 and Fujitsu VPP300) have reduced hardware costs by using DRAM rather than SRAM memory, and others (such as Fujitsu and NEC vector computers) have boosted arithmetic throughput by increasing functional unit parallelism, for example, by dividing the operations of a single vector instruction among multiple identical functional units. Our approach to hiding latency is to simulate many virtual PRAM processors per physical vector processor, such that each vector instruction simulates one step of the PRAM algorithm for up to $V_{\max}$ virtual processors.

support for irregular computation: On a single step of a PRAM algorithm, each processor can make a memory reference to an arbitrary location. Thus, simulating a single step of the algorithm requires vector instructions that can access memory using a vector of arbitrary addresses. In many cases, regular access patterns can be simulated with vector instructions that access a set of memory locations with a constant stride. But for simulating irregular accesses, it is critical to have hardware support for indirect vector loads and stores with *gather* and *scatter* instructions that generate addresses from an index vector. Gather and scatter instructions were not available on early vector machines such as the Cray-1 and TI ASC, nor on early models of the Cray X-MP, but were available in some memory-to-memory vector computers such as the CDC Star 100, Cyber-203, and ETA-10. These instructions have since become standard features in virtually all vector computers, including machines from Cray, Convex, Hitachi, Fujitsu, and NEC. Indirect addressing with gather and scatter is typically more expensive than with vector load and store, but far cheaper than scalar memory operations.

support for conditional computation: On a single step of a PRAM algorithm, a processor can perform conditional computation, such as finding the maximum of two elements. On a vector machine, conditional computation is performed using a *vector mask*, a special register with $V_{\max}$ bits. Vector test instructions set each bit based on a boolean test of each element of a vector register. The vector mask is used for a collection of instructions, including vector merge, an instruction that uses the vector mask to select operands from one of two vector registers on an element-by-element basis, and compress-index, an instruction that returns the index of each element of a vector register that passes a boolean test. Some machines also support conditional memory and arithmetic operations controlled by the vector mask.

simulating virtual processors with loop raking: A particular type of strided access frequently arises when using virtual processors. When simulating parallel algorithms that have sequential access patterns, we use a technique called *loop raking* [29] in order to preserve the order in which data elements are processed in parallel algorithms. This is essentially a vectorization technique, but the same technique is applicable to a pipelined microprocessor. The typical way to vectorize an array operation uses strip mining [156] to process a vector in contiguous blocks of v elements. In this method, each element of a vector-register, that is, each virtual processor, processes every vth element in the array. In order to give each virtual processor a contiguous block of elements, loop-raking uses a constant stride to access a set of elements evenly distributed across the input vector—as if a rake was placed over the vector, where each prong of the rake is a virtual processor. Since we are using a non-unit stride of $s = n/v$ to access an n-element array, we have to be careful about bank conflicts. On interleaved memory systems that use a cyclic distribution of consecutive words to memory banks, conflicts can be avoided by using a stride that is relatively prime to the number of banks. For example, if the number of banks is a power of two, v can be adjusted so that the stride, s, is odd. In a previous paper [48], we show how to handle cases when the array size n is not evenly divisible by v. To simplify the presentation, we omit these details in the remainder of the thesis.

3.2 Low-contention algorithms: a pragmatic approach

In this section, we first describe our goals in designing and adapting low-contention algorithms and contrast them with goals of theoretical work on contention. We then discuss three techniques for managing contention: randomization, replication, and direct data distribution for the (d, x)-BSP.

Theoretical work related to memory contention falls into two main areas: developing or adapting algorithms for low-contention models, and showing the relationships among models with different contention rules. Theoretical work on PRAM algorithms and simulations tends to emphasize asymptotic analysis, typically for arbitrary numbers of processors (and thus little data per processor). In contrast, we typically analyze constant factors, and we focus on modest numbers of processors and high slackness (typically between one and a few hundred processors, each with data set sizes in the thousands or millions).

For many fundamental problems, EREW (or QRQW) algorithms have been developed that match the asymptotic running time of the best CRCW algorithms. In some cases these algorithms are practical. Blelloch [23] showed several examples of converting CRCW algorithms to EREW algorithms using scan operations. However, for a given task, the best EREW algorithm is often more complicated than an equivalent CRCW algorithm, which can lead to higher constant factors as well as additional programming time.

More generally, CRCW algorithms can be optimally simulated on a BSP when there is sufficient slackness [85]. Unfortunately, general techniques for emulating arbitrary CRCW algorithms use expensive collective operations (such as integer sorting) on each step, and the existence of these general, yet somewhat impractical, simulation results removes much of the incentive for developing algorithms for cases where there is high slackness. Thus, most theoretical work on reducing contention focuses on cases where there are few elements per processor. In contrast, we typically consider cases with high slackness in which there is no need to incur the costs of general emulations.

The QRQW model helps bridge the gap between practical and theoretical concerns. Several algorithms and data structures (e.g., the binary search fat-tree described in Section 3.3.1) designed for the QRQW are fast in practice. The QRQW model itself gives us a way of reanalyzing CRCW algorithms, and in some cases we can determine that existing algorithms have tolerable contention (e.g., the dart-throwing random permutation algorithm described in Section 3.3.2). However, despite the practicality of the QRQW model, our goals are still somewhat different from those of theoretical studies. One distinction is that theoretical work is focused on very low contention algorithms (typically logarithmic in problem parameters), whereas, in practice, it is often possible to tolerate contention that is polynomial in problem size, for example, $O(\sqrt{n})$ or even linear in the problem size with a sufficiently small constant.

In the absence of a practical algorithm designed for the QRQW or EREW model, several heuristic techniques can be used to reduce contention. The approach we take in design is to analyze the constant factors and, in the implementation, to optimize for common cases. For example in using replication to reduce contention, we can tune the constant factors based on machine parameters (typically p and d) rather than developing an algorithm that eliminates contention entirely. If we are using a CRCW algorithm, we can be selective about using expensive emulation techniques. For example, it might be advantageous to sample memory access patterns to estimate an upper bound on the maximum contention in order to select an appropriate emulation method.

In our experience, designing programs using the QRQW is adequate for avoiding bank contention. However, when tuning an application, it can be beneficial to look at a more detailed model, for both regular and irregular access patterns. In these cases we can implement algorithms or portions of algorithms (e.g., accesses to particular data structures) directly for the $(d, \mathbf{x})$-BSP.

3.2.1 Randomization

Randomization is useful for designing low-contention algorithms [87], and as discussed in Chapter 2, for reducing module map contention and network contention. As with previous work, we assumed in Chapter 2 that the memory locations are hashed to memory banks using a truly random mapping. In practice, however, the mapping cannot be truly random, since it should be efficiently computable for every memory address. Thus, it is important to consider the implementation of randomization, in hardware or software.

Hardware support for fast computation of pseudo-random hashing is provided on some machines, including the TERA MTA and the Fujitsu μ-VP (used on nodes of the Meiko CS-2). On machines that do not provide hardware-supported hashing (such as the CRAY vector machines), general PRAM simulation using random hashing could incur significant overheads, including the cost of computing hash functions, the cost of additional indirection, and increased temporary memory requirements. For some algorithms, however, it is possible to approximate the effect of randomization by randomly permuting the input and some of the intermediate results. (A QRQW algorithm for random permutation is described later in Section 3.3.2.) In others, the nature of the algorithm results in random mapping without any additional steps. Even for algorithms that require randomization, computing a pseudo-random hash in software is not prohibitive, particularly if applied only to data structures that are likely to be sources of contention. The evaluation costs for a variety of hash functions on the CRAY C90 are given in Table 3.1.

When hash functions are used for pseudo-random mapping of memory locations to memory banks, it is important that they exhibit favorable properties for any given input (i.e., that they are "universal"). The function h_a^1, which is called the *multiplicative hashing scheme* in [121, p. 509], was recently shown by Dietzfelbinger et al. [69] to be 2-universal in the sense of Carter and Wegman [46]: for any two distinct numbers $x, y \in [0..2^u - 1]$, $\mathbf{Prob}\,(h_a^1(x) = h_a^1(y)) \leq 1/2^{m-1}$; i.e., the collision probability is approximately the same as for a random mapping. The actual choice of a hash function may be influenced by several factors, including its degree of universality, its evaluation cost, and its congestion behavior, both theoretically (see [68]) and experimentally (see [73]).

3.2.2 Replication

Replication is a well-known strategy for reducing contention. In the context of implementing PRAM algorithms for the (d, x)-BSP, replication is useful primarily for implementing CRCW algorithms, and secondarily for reducing module map contention. General techniques for emulating CRCW algorithms have been developed [85], but in some cases, replication is a more practical alternative than general emulations based on sorting. Replication can be applied in a variety

Hash Function	T/n
Linear	
$h^1_{a,b}(x) = ((ax + b) \bmod 2^u) \text{ div } 2^{u-m}$	1.8
$h^1_a(x) = (ax \bmod 2^u) \text{ div } 2^{u-m}$	1.8
Quadratic	
$h^2_{a,b,c}(x) = ((ax^2 + bx + c) \bmod 2^u) \text{ div } 2^{u-m}$	3.4
$h^2_{a,b}(x) = ((ax^2 + bx) \bmod 2^u) \text{ div } 2^{u-m}$	3.4
Cubic	
$h^3_{a,b,c,d}(x) = ((ax^3 + bx^2 + cx + d) \bmod 2^u) \text{ div } 2^{u-m}$	6.7
$h^3_{a,b,c}(x) = ((ax^3 + bx^2 + cx) \bmod 2^u) \text{ div } 2^{u-m}$	6.7

Table 3.1: The evaluation cost of software implementations of various hash functions in terms of clock cycles per element (for each CRAY C90 processor). The functions map items x from the domain $[0..2^u]$ into the range $[0..2^m]$, and a, b, c, and d are odd numbers selected at random from $[1..2^u - 1]$.

of ways, ranging from simple schemes that replicate an entire data structure to more intricate schemes that replicate portions of a data structure based on expected access patterns.

In designing a strategy for replication, there are tradeoffs among three components: the time to measure or estimate the contention, the time to replicate data and combine updated data, and the time lost due to contention. In some applications, there is no need to measure the contention because it is a known property of the algorithm, whereas in other applications, the cost of estimating contention would be prohibitive.

To illustrate these tradeoffs, we consider the task of simulating a single step of a CRCW algorithm, where there are n elements accessed from an array of size m, and where the location of the n elements is unknown. We first analyze the running time using a simplified version of the (d, x)-BSP model, where module map contention is ignored, the gap g is 1, and the delay d is normalized to the gap. This formulation is equivalent to a QRQW model where the penalty for memory contention k is $f(k) = k \cdot d$.

We describe a read step that gathers elements from a source array. (Simulating a write step has the same complexity.) The worst-case performance can be reduced by replicating the source array. The cost of replicating the entire array r times is simply $O(r \cdot m/p)$. If the gather is executed by randomly selecting among copies, and if the contention is high enough to spread accesses among copies fairly evenly, then the cost of the gather will be approximately $O(\max(n/p, k/r \cdot d))$, where $k = n$ in the worst case.

The tradeoff between the cost of the replication and the cost of the gather is illustrated in Figure 3.1. Note that most of the benefit comes from the first few copies; if the replicated array is relatively small (i.e., if the ratio n/m is high), the first n/m copies are fairly inexpensive.

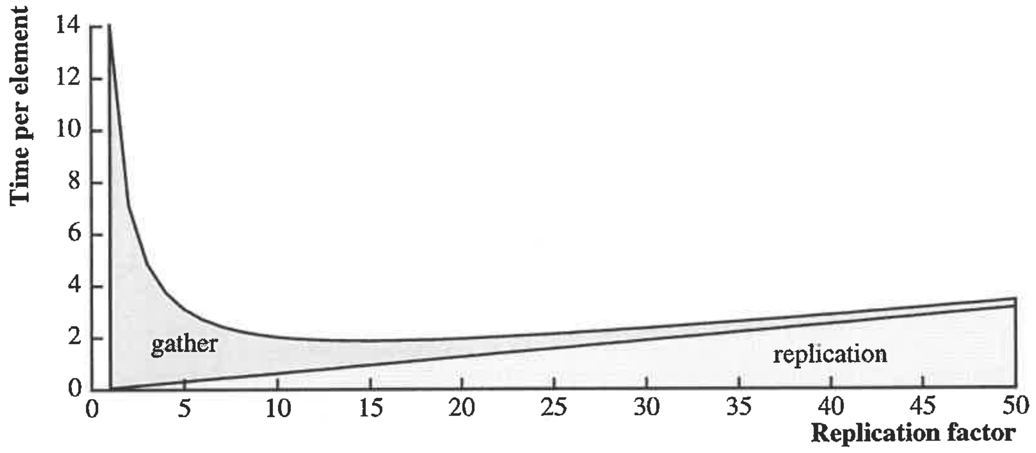


Figure 3.1: Tradeoff between replication and contention. The curves show predictions of the worst-case time per memory access where the ratio of memory accesses to source array elements (n/m) is 4. The lighter shaded area represents the replication cost, and the darker shaded area represents the memory access cost. The top curve represents the sum of these components. The machine parameters used to generate the predictions were measured on an 8-processor CRAY J90.

3.2.3 Data distribution

On machines without random hashing in hardware, we need to account for the module contention in the mapping of locations to memory banks. One way to avoid module contention is to design algorithms for the (d, x)-BSP memory organization. This approach is similar to designing *direct* BSP algorithms [85], in which portions of data structures are explicitly assigned to memory modules rather than randomly mapped to modules. It is also similar to the notion of data distribution used in parallel languages such as High Performance Fortran (HPF). Although direct BSP algorithms and HPF data distributions are both distributed memory models, many of the same concepts can be applied to a model based on a shared memory composed of many memory banks.

On vector multiprocessors, low-order interleaving of addresses to memory banks is commonly used. The resulting cyclic distributions work well for one-dimensional vectors, but do not always work well for multi-dimensional or irregular data. Several techniques are commonly used for avoiding bank conflicts on regular data structures, including padding leading dimensions of multi-dimensional arrays, accessing multi-dimensional arrays along diagonals, and redistributing data across banks (e.g., in an FFT [14]).

Distributing irregular data is more difficult, primarily because the reference patterns are not known (in general) until run-time. HPF extensions to accommodate irregular data have been proposed [190, 160], and data distribution for irregular data structures is an area of active research [102, 118, 123, 136, 174, 176]. However, data distribution techniques for irregular data structures have not been used widely on interleaved memory systems. In the (d, x)-BSP, even when there is no memory location contention, data-dependent addresses can still cluster into particular banks. In some cases this type of memory bank conflict can be avoided using appropriate data distributions. In particular, if each processor (or virtual processor) issues irregular accesses to disjoint portions of a data structure, these portions can be distributed across banks. This technique is illustrated in Chapter 4 in the context of a histogram operation, in which the "local" memory used by each virtual processor is assigned to a different bank, or small set of banks.

3.3 Basic algorithms

This section describes algorithms for binary search, random permutation, and scan primitives. These basic algorithms serve as building blocks for designing other algorithms and demonstrate several concepts and techniques described in this chapter.

3.3.1 Binary search

We consider a simple QRQW binary search algorithm for looking up n keys in an ordered collection of m elements [87]. Such binary searching is an important substep in several algorithms for sorting and merging (e.g., [158]). A natural data structure for the CRCW PRAM is a balanced binary search tree. However, this data structure is inadequate when accounting for contention, since simultaneous searches in a binary tree will cause high contention near the root.

The contention can be reduced in several ways. One way is to use an EREW algorithm for the scan-vector model [23]. The algorithm works by maintaining a vector of pointers into the search tree. Initially there is a single vector of elements pointing to the root, and at each level of the search, vectors of pointers are partitioned according the direction followed in the search tree. Concurrent reads are replaced with segmented broadcast operations (also called segmented copy scans). A second option is to perform the binary searches in a sorted array, beginning the binary search for each processor at a randomly selected starting location. This will avoid contention in the early iterations of the binary searches. A third option, which we describe in more detail, is to use a data structure with inherent replication, a *binary search fat tree*, as depicted in Figure 3.2.

The search algorithm replicates nodes of the search tree to avoid contention, and at each level, selects one of the replicated nodes at random. This data structure is interesting from the point of view of the QRQW PRAM and the (d, x)-BSP since the amount of replication needed will depend on the contribution of the contention term to the running time, and in general will

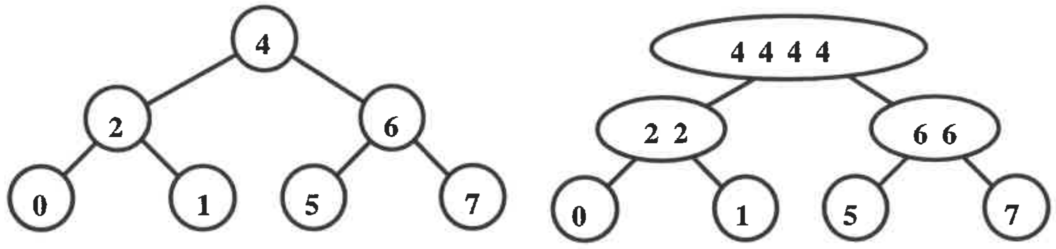


Figure 3.2: A binary search tree and a "fattened" binary search tree.

present a tradeoff.

The (d, x)-BSP model can predict the running time of the binary search for different amounts of replication. In our implementation experiment, the root is replicated f times (the "fatness"), and each level below the root is replicated half as many times as the level above. Thus, there are $\max(2^i, f)$ nodes at level i of the fattened tree. For simplicity, we consider the case where the number of nodes at each level is less than the number of banks. Assuming an equal number of lookups to each key, the expected time per lookup is:

$$\sum_{i=0}^{\lceil \lg m \rceil - 1} \max\left(\frac{cg \cdot S}{n}, \frac{d \cdot \mathbf{E}(R)}{n}\right) = \sum_{i=0}^{\lceil \lg m \rceil - 1} \max\left(\frac{cg}{p}, \frac{d}{\max(2^i, f)}\right) ,$$

where the implementation-dependent constant c is approximately 3.9 on the CRAY J90 and 1.4 on the C90*. (Recall that S is the maximum number of requests made by any processor and R maximum number of requests at any memory. $\mathbf{E}(R)$ denotes the expected value of R.) Figure 3.3 shows that the predictions are accurate, underestimating the running time by at most 20 percent.

3.3.2 Random permutation

Another example of a QRQW algorithm is a "dart-throwing" algorithm for generating random permutations. This algorithm first generates n random indices in the range $[0..cn]$ for some constant c ($c = 2$ in our experiments). Each element i then writes its self-index into a destination array at the location specified by the ith random index. Elements for which there are collisions repeat another round. The rounds continue until no elements remain. At this point, the values written into the destination array are packed into contiguous locations, producing the index for the random permutation. The algorithm runs in $O(n/p + \lg n)$ time on a QRQW PRAM.

The predicted running time for the QRQW dart throwing algorithm can be derived from experimentally measured arithmetic computation costs, the number of global references (using gather or scatter) and the expected total number of darts thrown. The time is linear in the

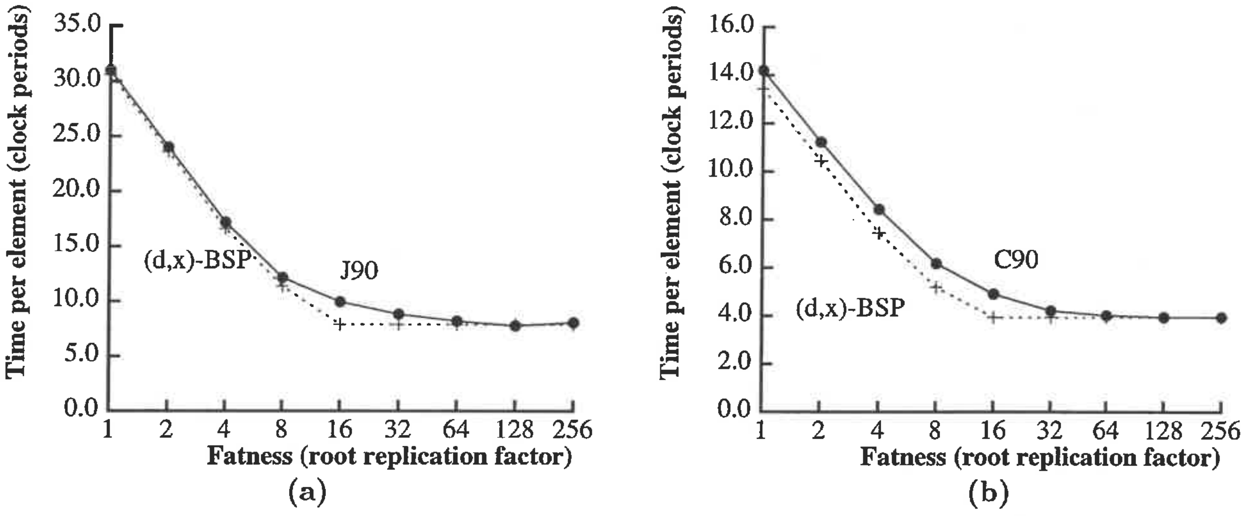


Figure 3.3: Predicted and measured times for a binary search fat-tree algorithm, as a function of the "fatness" of the tree. (Each search tree has 256 unique elements.) The dotted curves show predicted times and the solid curves show measured times on 8 processors of the (a) CRAY J90 and (b) CRAY C90.

problem size and is given by:

$$\begin{aligned} t_{QRQW} &= 200000 + (18.7 + 4 \cdot g + w \cdot (5 \cdot g + 12.3)) \cdot n/p \\ &= 200000 + (37.8 + 11.8 \cdot g) \cdot n/p \qquad (3.1) \end{aligned}$$

where $w = 1.55$ is the ratio of total darts thrown to the permutation size. There is less than 1 percent variation in w as n is varied.

It is interesting to compare the running time of this algorithm to that of an EREW algorithm (a sorting-based algorithm, which is the most practical EREW algorithm in the literature, to the best of our knowledge). The EREW algorithm is based on a radix sort (described in Chapter 4) that ranks $2.5 \cdot \lg n$ bits and checks for duplicate keys. Using a performance model for radix-sort adapted from Chapter 4, the predicted running time for the EREW algorithm is given by:

$$t_{EREW} = 400000 + \left\lceil \frac{2.5 \cdot \lg n}{\lg(n/p) - 8} \right\rceil \cdot (3.5 + 7 \cdot g) \cdot n/p \qquad (3.2)$$

where the 2.5 comes from the number of bits, 8 comes from latency hiding ($\lg(128) + 1$), 3.5 comes from arithmetic on buckets, and 7 comes from indirect operations for histogram and permutation.

Results of the experiments are shown in Figure 3.4. (A similar experiment, on the MasPar MP-1, was reported in [87]; for the EREW algorithm, the system sort was used.) This experiment illustrates that allowing a controlled amount of contention enables the use of a more practical algorithm.

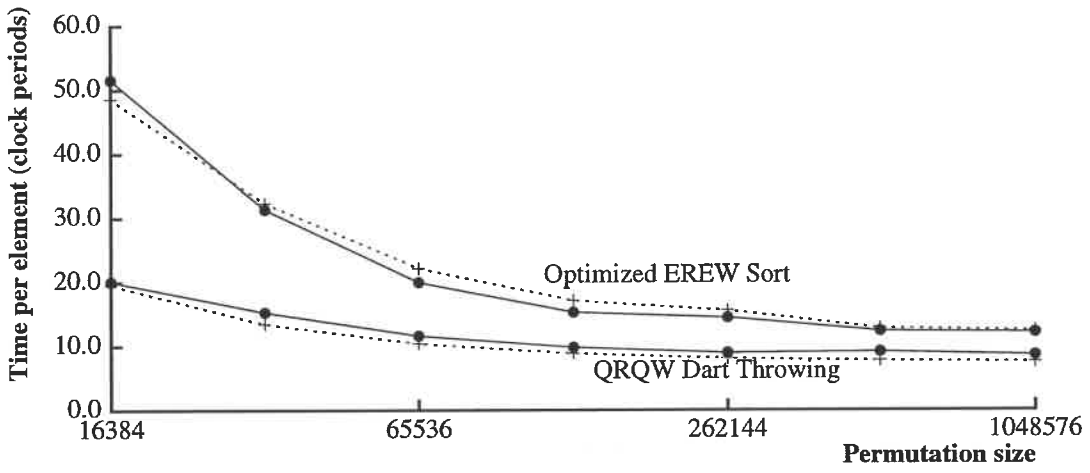


Figure 3.4: Performance on an 8-processor CRAY J90 for two algorithms that generate a random permutation: a QRQW dart-throwing algorithm and an EREW sorting-based algorithm. The dart-throwing algorithm generates large permutations in under 10 clock periods per element. The predicted times, shown with dotted curves, are given by equations 3.1 and 3.2. (The timings do not include the time for random number generation; however, the two algorithms require approximately the same number of random bits.)

3.3.3 Scan primitives

Scans are powerful primitives for designing parallel algorithms [22, 126], particularly for solving problems that require load-balancing or manipulating irregular data structures. They are also useful for converting concurrent memory access to exclusive memory access [23]. Several collection-oriented languages, including NESL, APL, and HPF, provide scan primitives.

The implementation of scans provides a simple example of the techniques used for hiding memory (and functional unit) latency. In this section, we describe the implementation and applications of scan operations. The implementation techniques for vector machines were developed jointly with Sid Chatterjee and Guy Blelloch [48, 29].

The *scan operation*, also called the all-prefix-sums computation, takes a binary operator $\oplus$, and an array $[a_1, a_2, \ldots, a_n]$ of n elements, and calculates the values x_i such that

$$x_i = \begin{cases} a_1, & i = 1 \\ x_{i-1} \oplus a_i, & 1 < i \leq n \end{cases}$$

If $\oplus$ is associative, the scan can be calculated efficiently in parallel in $O(\lg n)$ time [122, 129].

We first describe a parallel algorithm for a simple plus-scan operation (i.e., where $\oplus$ is integer addition) and describe the implementation and analysis for pipelined-memory multiprocessors. Section 3.3.3.2 describes several extensions to this scan algorithm.

3.3.3.1 Scan implementation

We first review a simple parallel algorithm for performing a plus scan on an array of length n on p processors. It has three phases:

1. *Local sum:* Processor i sums elements is through $(i+1)s-1$, where $s = n/p$. (For simplicity, we assume that n is a multiple of p.) Call this result Sum[i].

2. *Cross-processor scan:* The Sum array is now scanned.

3. *Local scan:* Processor i sums its portion of the array, starting with Sum[i] as the initial value, and accumulating the desired partial sums.

In a PRAM model, the running time of the first and third steps is simply $O(n/p)$. The second step, the scan across processors, runs in $O(\lg p)$ for a tree-summing algorithm [129], or $O(p)$ for a serial algorithm. The scan algorithm is work-efficient for $n = O(p \lg p)$, or if the serial cross-processor scan is used, for $n = O(p^2)$.

To implement a scan on a pipelined-memory multiprocessor, we simulate the algorithm for $p \cdot v$ virtual processors. On a vector machine, we use loop raking (as described in Section 3.1.3) to simulate the access pattern of the virtual processors. The second step, the cross-processor scan, can be implemented in several ways, for example, with a serial scan, with a parallel tree-reduction, or with a hierarchical implementation that combines a serial scan within processors with a tree-reduction across processors. The selection of the best implementation depends on the synchronization cost L, as well as the number of the processors and input size.

3.3.3.2 Generalizations

This section describes three generalizations of the scan algorithm in Section 3.3.3.1.

segmented scans: The *segmented* scan operations take an array of values and a data structure specifying how this array is partitioned into segments [22]. A scan is executed independently within each segment. As with the unsegmented version, the segmented scans can also be implemented in parallel in $O(\lg n)$ time [22]. Segmented scans can be implemented efficiently on vector computers [48] using vector merge instructions to handle segment boundaries and loop raking to access the data [29].

Segmented scans were originally introduced to solve problems with irregular data structures on the Connection Machine CM-2. Segmented scans can be used to implement many data-parallel algorithms for problems with irregular data structures, including sparse matrix routines [22, 23], parallel quicksort [22], and machine learning [36], computer graphics [60], object recognition [197], processing image contours [50], and network optimization [150]. Because of their usefulness for such problems, hardware support for segmented scans was included in the Connection Machine CM-5 [133], and scan intrinsics are included in HPF.

Segmented scans can be implemented in several ways, due to three sources of parallelism: the scans over different segments are independent; each scan over each segment is parallelizable; and a segmented scan can be expressed as a single parallelizable linear recurrence [25]. The main benefit of implementing segmented scans in terms of recurrences is that latency is hidden regardless of the segmentation. Other approaches, such as parallelizing only within segments, or only across segments, are more sensitive to the locations of the segment boundaries. Chapter 6 compares several of these alternatives in the context of sparse matrix multiplication.

linear recurrences: An m^{th}-order linear recurrence is a set of equations of the form:

$$x_i = \begin{cases} c_i, & 1 \le i \le m \\ (x_{i-1} \otimes a_{i,1}) \oplus \cdots \oplus (x_{i-m} \otimes a_{i,m}) \oplus b_i, & m < i \le n \end{cases}$$

where $\oplus$ and $\otimes$ are binary associative operators, and $\otimes$ distributes over $\oplus$. Such linear recurrences are used in scientific applications [127], in the design of parallel algorithms [126, 22], and in solving a broader class of recurrences [122, 79].

In [29], we present a variation of the partition method for solving linear recurrences. The algorithm is based on the scan algorithm presented earlier and uses loop raking. As compared to the more commonly used version of the partition method for linear recurrences, the algorithm reduces the number of memory accesses and temporary memory requirements, and it is better suited for a multiprocessor implementation because it uses less synchronization and is less sensitive to latency.

compaction: Given a vector of values a vector of boolean flags, a *linear compaction* (also referred to as a *pack* operation) produces a vector the input values with corresponding *true* flags. Compaction can be expressed in terms of scan operations as a straightforward combination of a scan to enumerate the final location of the true elements, followed by a permutation to place the true elements in these locations.

The pack operation has a number of applications, including reducing work, load-balancing, implementing conditionals, and flattening nested parallelism [23]. Compaction is also useful for implementing parallel while loops, for example, by extracting the active elements after each iteration. This technique has been used for vectorizing several classes of algorithms including particle codes [137, 42], tree traversals [140], hashing [115], and list ranking [164].

An example of an algorithm that uses packing for both theoretical and practical improvements is the quickselect algorithm [105]. This algorithm finds the kth smallest element of a set by recursively partitioning the set based on a pivot value and retaining only the subset

primitive	t_e unsegmented	t_e segmented
vector add	1.1	N/A
first-order linear recurrence	1.8	N/A
gather	1.8	2.1
scatter	2.1	2.6
plus-reduce	0.6	1.3
plus-scan	1.4	1.9
pack	1.7	2.6

Table 3.2: Times for vector primitives, expressed in clock cycles per data element, on a single processor of the CRAY C90, Gather and scatter operations were timed with random permutations. Segmented reduction was timed on a segmented vector with an average segment length of 10. Pack was timed on a vector of elements in which elements to be compacted were selected independently with a 50 percent probability.

that contains the pivot. The pack operation implicitly balances the load after each iteration, in addition to reducing the amount of work. (Timings for quickselect on various machines are given in an evaluation of the implementation of NESL [27].)

3.3.3.3 Scans in language implementation

Many of the applications of scans mentioned so far can be generalized and applied automatically by a compiler. To explore this idea, we implemented the back end support for running NESL on one processor of the CRAY C90 and J90. The NESL compiler makes extensive use of scan and pack operations in the code it generates. In addition to supporting explicit scan operations with NESL operators, the compiler uses scans and packing when performing transformations to *flatten* nested parallelism [23].

The back end support consists of a C Vector Library (CVL) [26], with a variety of elemen-twise operations and segmented vector operations (e.g., sum, scan, permute, pack, split). The CVL operations are implemented in Cray Assembly Language using a custom macro-assembler that includes a set of macros for programming with virtual processors. The CVL library is used a back end for NESL [24] via an interpreter for the VCODE [28] intermediate language. The CRAY implementation runs nested primitives well (typically at less than twice the cost of the corresponding unsegmented primitives) and supports EREW PRAM algorithms well (with permutation operations that are only a factor of two slower than elementwise instructions on the CRAY C90 and CRAY J90). Table 3.2 summarizes the performance on the CRAY C90 of several types of reductions, scans, recurrences, and compares their performance to other vector operations. The performance of this system on a few benchmarks is reported in [27].

3.4 Discussion

This chapter presented several techniques and basic algorithms for developing efficient implementation of irregular algorithms on pipelined-memory multiprocessors. These techniques will be applied in algorithm experiments described in later chapters. In particular, tradeoffs between work and slackness are explored in the chapters on sorting and merging. Contention is discussed in all the algorithm experiments, with explicit data distribution highlighted in the context of radix sorting, and with randomization techniques highlighted in the context of connected components. Binary search is discussed for merging, and scan primitives are used in several chapters: unsegmented scans for radix sort, segmented scans for sparse matrix multiplication, and compaction for connected components.

Chapter 4

Sorting

This chapter describes the efficient implementation of a parallel radix sort algorithm for high-bandwidth pipelined-memory multiprocessors. As with all the algorithms described in the thesis, latency is tolerated by simulating multiple PRAM virtual processors on each physical processor. In contrast to most of the algorithms described in the thesis, all the simultaneous memory accesses in the radix sorting algorithm are exclusive. Thus, we use this case study as an opportunity to explore the issue of module map contention, that is, the performance impact of various mappings of memory locations in the PRAM algorithm to physical memory modules of the $(d, \mathbf{x})$-BSP.

The chapter is organized as follows: Section 4.1 explains the advantages of radix sort over other sorting algorithms and describes previous work on vectorized sorting. Section 4.2 introduces serial and parallel algorithms for radix sort. Section 4.3 describes implementation issues for radix sort on high bandwidth multiprocessors. In particular, it describes techniques for tolerating latency on pipelined-memory multiprocessors and it describes how to select an appropriate mapping of memory locations in the PRAM algorithm to memory modules. Section 4.4 introduces a performance model for the CRAY C90 relating the running time of the radix sort to various machine and problem parameters. The model is used to select an optimal radix, analyze latency hiding, and accurately predict running times over a range of data sizes and numbers of processors. Section 4.5 summarizes the implementation results. The CRAY C90 implementation of a PRAM radix sort algorithm is stable, predictable, insensitive to the distribution of keys, and fast over a wide range of problem parameters. The implementation for a 16-processor CRAY C90 sorts at a rate of over 100 million 64-bit keys per second.

This work was performed jointly with Guy Blelloch and was originally published in a conference paper that reported performance on the CRAY Y-MP [207].

4.1 Background and previous work

In order to justify our choice of radix sort, we briefly present some background material on practical parallel sorting and review previous vectorized and parallelized sorting implementations. Further background can be found in surveys of parallel sorting [4, 169], in comparisons of practical sorting algorithms [34, 30, 103, 67], and in studies of sorting algorithms for the BSP model [85, 83, 180, 84].

Several issues guide the choice of an algorithm for a practical sorting implementation [30]. In most cases, the key issue is speed, which is determined primarily by the amount of work performed and the match of the algorithm to the architecture. Other significant issues include space efficiency, the ability to handle arbitrary distributions of keys, determinism, stability (i.e., control of the output order for equal keys), and generality of the interface (i.e., the ability to sort complex records and the ability to both rank and sort keys).

Radix sorting is an attractive alternative to comparison-based sorts, such as quicksort [106], since for n fixed-length keys, its running time is $O(n)$ instead of $O(n \lg n)$ time. This is a significant savings in theory, and because of small constant factors in the running time of radix sort, a significant savings in practice.

However, there are two potential disadvantages of radix sort. One potential disadvantage is that radix sort cannot use arbitrary comparison functions. To use a radix sort, it must be possible to break the key into digits. As a practical matter, this does not appear to be a drawback. Sorting of common types of keys, such as floating-point numbers, integers, or character strings (based on lexical order), can be implemented efficiently with radix-based sorts. Furthermore, the fastest implementations of comparison-based sorts are often coded such that they cannot take a user supplied comparison function, and typically sort only integers or floating-point numbers.

Another potential disadvantage is that radix sort accesses memory in an arbitrary order. Therefore, on a machine with a cache, many memory references will cause cache misses and, potentially, TLB misses and page faults. Studies have shown that on serial machines, when the problem fits into physical memory but not into the cache, radix sort performs no faster, or only marginally faster, than optimized implementations of quicksort (for 32-bit and 64-bit integers) [146, Chapter 8]. The random access patterns in radix sort, while clearly an important issue on cache-based machines, are not a concern for high-bandwidth multiprocessors with efficient support for fine-grained memory access.

However, the choice of an algorithm for a high-bandwidth pipelined-memory multiprocessor is guided by two additional issues, namely the ability to hide latency and minimize memory bank contention—without introducing an excessive amount of additional work. Previous work on sorting for vector computers has failed to meet these objectives simultaneously. Several implementations have used highly inefficient algorithms that perform $O(n^2)$ work, including implementations of bubble sort [170], insertion sort [53], and odd-even transposition sort [170, 154]. Implementations of $O(n \lg^2 n)$ work parallel algorithms, including implementations of merge-exchange sort [187, 170, 45], bitonic sort [155], diamond sort [39, 170, 45, 147], and shellsort [53], are considerably faster than the $O(n^2)$ algorithms, but typically are not competitive with an efficient serial algorithm. On the other hand, most implementations of $O(n \lg n)$ work algorithms, including several implementations of quicksort[187, 170, 45, 70, 154] and an implementation of heapsort [170] have sacrificed latency-hiding (i.e., vectorization) in order to use an efficient algorithm. Latency-tolerant implementations of quicksort [187, 135, 170, 70] have been developed, although, to our knowledge, no implementation of quicksort has been effectively vectorized and parallelized. There have been other implementations of radix sort for vector computers including an implementation of scalar radix sort [53], and implementations of binary radix sort [70, 48]. The latter implementations are vectorized, but at the expense of requiring one pass over the data for each bit of the sort key, an increase in work that negates the advantage of radix sort.

Our conclusion in reviewing previous work is that radix sort is likely to be the fastest algorithm if we can find an effective implementation strategy for high-bandwidth machines.

4.2 Radix sort

4.2.1 Serial radix sort

We first review the serial radix sort algorithm since our parallel algorithm will include the same phases (see [56, Section 9.3] for more details). Radix-based sorting algorithms treat b-bit keys as multidigit numbers in which each digit is an integer with a value in the range $\langle 0..(m-1)\rangle$, where m is the radix. A 32-bit integer, for example, could be treated as a 4-digit number with radix $m = 2^{32/4} = 2^8 = 256$. The radix m is usually chosen to minimize the running time and is highly dependent on the implementation and the number of keys being sorted.

Radix sort works by breaking keys into digits and sorting one digit at a time, starting with the *least* significant digit. A *counting sort* (sometimes called distribution sort or bucket sort) is used to sort each digit. Since radix sort starts from the least significant digit, the sort works only if the ordering generated in previous passes is preserved. Each counting sort therefore must be stable.

The counting sort is implemented as follows. We assume that each digit consists of r bits. The basic idea is to determine, for each input digit $D[i]$, the number of keys with a digit less than $D[i]$, and the number of keys with a digit equal to $D[i]$ appearing earlier in the input sequence. To do this, the sort uses $m = 2^r$ buckets, one for each possible digit value.

The following pseudo-code sorts an array of keys K, based on an array of r-bit digits D and places the result in R. The counting sort algorithm is divided into 3 phases:

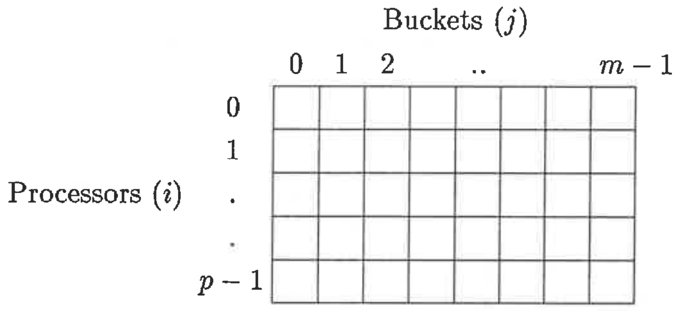
```
HISTOGRAM-KEYS
  do $i \leftarrow 0$ to $2^r - 1$
    $Bucket[i] \leftarrow 0$
  do $j \leftarrow 0$ to $n - 1$
    $Bucket[D[j]] \leftarrow Bucket[D[j]] + 1$
SCAN-BUCKETS
  $Sum \leftarrow 0$
  do $i \leftarrow 0$ to $2^r - 1$
    $Val \leftarrow Bucket[i]$
    $Bucket[i] \leftarrow Sum$
    $Sum \leftarrow Sum + Val$
RANK-AND-PERMUTE
  do $j \leftarrow 0$ to $n - 1$
    $A \leftarrow Bucket[D[j]]$
    $R[A] \leftarrow K[j]$
    $Bucket[D[j]] \leftarrow A + 1$
```

The first loop of HISTOGRAM-KEYS clears the buckets. The second loop, at iteration j, uses the j^{th} element of the digit array as an offset into the buckets, and increments the count in that bucket. At the end of HISTOGRAM-KEYS, $Bucket[i]$ contains the number of digits having value i. SCAN-BUCKETS performs a *scan* operation on the buckets, returning to each element the sum of all previous elements. After the scan, $Bucket[i]$ contains the number of digits with a value j such that $j < i$. This quantity is the position in the output in which the first key with digit i belongs. In the final phase, RANK-AND-PERMUTE, each key with a digit of value i is placed in its final location by getting the offset from $Bucket[i]$ and incrementing the bucket so that the next key with digit i gets placed in the next location. Since the keys are processed in increasing order, COUNTING-SORT is stable.

4.2.2 Parallel radix sort

The serial algorithm cannot be directly parallelized (or vectorized) because of loop dependences in all three phases. If an attempt is made to parallelize over the iterations of the histogram computation, several processors could attempt to increment the same bucket simultaneously. If the bucket is not locked by each processor to gain exclusive access, only one of the increments would take effect. On the other hand, if the bucket is locked, the bucket would act as a serial bottleneck and would greatly degrade performance if many keys had the same digit. It is possible to parallelize the algorithm without requiring a lock on the buckets by using a separate set of buckets for each processor. Each processor processes n/p of the keys and tabulates their distribution in its own set of buckets.

In this parallel version of radix sort for an EREW PRAM, the buckets can be viewed as a two-dimensional matrix *Buckets*$[i, j]$:



where i is the processor number and j is the bucket number. In the first and third phases of the parallel radix sort (HISTOGRAM-KEYS and RANK-AND-PERMUTE) each processor works on its own set of keys with its own set of buckets, thereby removing all dependences. The bucket scan, however, must be modified to combine the buckets from the different processors. If each processor scanned only its own buckets, the digits would be sorted within the processor but not across the whole input data. After the scan, *Buckets*$[i, j]$ should contain the position in the output where the first key from processor i with digit j belongs. This condition can be expressed as:

$$\mathit{Buckets}[i,j]_{\text{after}} \leftarrow \sum_{k=0}^{p-1}\sum_{m=0}^{j-1} \mathit{Buckets}[k,m] + \sum_{k=0}^{i-1} \mathit{Buckets}[k,j] \; .$$

That is, the offset is the total number of digits less than j over all the processors ($0 \le k < p$), plus the number of digits equal to j in processors less than i. This sum can be calculated by flattening the matrix into column-major order and executing a scan on the flattened matrix. The scan operation can be parallelized using a tree-summing or similar algorithm, as discussed in Section 3.3.3. The parallel version generates the same permutation as the serial algorithm, thus preserving stability.

A similar sorting algorithm was described by Johnnson [114], and a restricted version of this algorithm was described by Cole and Vishkin as part of an optimal 2-ruling set algorithm [54]. The full algorithm was also implemented on the Connection Machines CM-2 [34] and CM-5 [195].

4.3 Implementation for high bandwidth multiprocessors

In order to use the EREW PRAM radix sort algorithm for developing an efficient implementation, we need to find an efficient mapping of the algorithm onto high-bandwidth multiprocessors.

Since this is the first detailed case study in the thesis, we use it to make latency tolerating techniques more concrete. We first describe how the concept of virtual processors is used to implement a latency-tolerant histogram operation for various types of architectures. We then describe avoiding contention, and in particular, the performance impact of various mappings of buckets in the radix sort algorithm to memory banks on an interleaved memory system.

4.3.1 Hiding latency

As discussed in Section 3.1, we use the concept of *virtual processors* to hide latency. That is we simulate multiple PRAM processors per physical processor in order to pipeline the global memory access. In our parallel radix sort algorithm, there are two basic types of global random access: permutations of keys, and histogram (and ranking) operations that update buckets. For large sets of keys, hiding latency on the key permutation operations is straightforward, but efficiently hiding latency for the bucket update operations is slightly more complicated. In this section, we describe how to hide latency on random accesses to buckets. Later, Section 4.4 discusses some tradeoffs between algorithm and machine parameters.

We now describe the mapping of a parallel histogram onto various classes of pipelined-memory architectures. For each architecture, we show pseudo-code for a parallel histogram algorithm that hides memory latency. The structure of the code varies somewhat with the architecture, but in each case, we use the same algorithm and the same basic approach to hiding latency.

In the pseudo-code segments below, n is the number of keys, p is the number of physical processors, and v is a slackness parameter that denotes the ratio of virtual to physical processors (i.e., there are $v \cdot p$ virtual processors). The `key` array holds n keys and the two-dimensional `bucket` array holds one set of keys per virtual processor. The pseudo-code assumes that buckets have been initialized to zero and that all integer divisions have no remainder. The variables `self` and `index` are local to each processor; `self` identifies a (virtual) processor index and `index` identifies the key currently being examined by a processor.

A EREW PRAM algorithm for histogram using p sets of buckets can be expressed as follows:

```
s = n/p
do parallel j = 0, p-1
  self = j
  do i = 0, s-1
    index = j * s + i
    bucket(key(index), self) += 1
  end do
end do parallel
```

Each iteration of the outer loop is assigned to a different processor, and the inner loop tabulates a histogram of a contiguous block of n/p keys.

On a multithreaded machine that automatically assigns parallel loop iterations to processors, latency would be hidden simply by using v virtual processors per physical processor. The code would be nearly identical to the PRAM pseudo-code above, except that the number of processors, p, would be replaced by the number of virtual processors $p' = p \cdot v$.

If the iterations were statically scheduled in a block fashion for a multithreaded machine, the code could be structured as follows:

```
s = n/(v*p)
do parallel j = 0, p-1
  do multithread k = 0, v-1
    self = j*v + k
    do i = 0, s-1
      index = s * (k + v*j) + i
      bucket(key(index), self) += 1
    end do
  end do multithread
end do parallel
```

The `do multithread` block loops over the virtual processors assigned to a given physical processor. Notice that the total amount of work is still n bucket increments.

Interestingly, the vector code is similar to the above code for multithreading. A simple loop interchange enables a compiler to vectorize the innermost loop over virtual processors:

```
s = n/(v*p)
do parallel j = 0, p - 1
  do i = 0, s - 1
     do vector k = 0, v-1
       index = s * (k + v*j) + i
       self = j*v + k
       bucket(key(index),self) += 1
     end do vector
  end do
end do parallel
```

A vectorizing compiler converts the inner loop to vector instructions, namely a vector load of v keys, a gather of v bucket values, an increment of the v values, and a scatter of the incremented values. Notice that access pattern to the keys uses loop raking (introduced in Section 3.1.3), ensuring that each virtual processor operates on a contiguous block of keys. (This is not important for a simple histogram operation since the result is independent of the order of bucket increments, but is critical for the related RANK-AND-PERMUTE operation, which must preserve a stable ordering in order to implement radix sort correctly.)

Techniques similar to vectorization can be used for implementing the histogram on a pipelined-memory microprocessor, and in fact, the code structure would be similar to that of the vector code. With an appropriate choice of the number of virtual processors, the inner (vector) loop could be fully unrolled, and the memory accesses could be scheduled to allow for latency tolerance. For example, with a non-blocking cache, all the bucket loads could be issued in a burst (rather than issuing sequential load-increment-store sequences), so that cache misses to the buckets would be overlapped.

For a simple histogram, using virtual processors might not be necessary on some microprocessor architectures, depending on the specific mechanisms used for latency hiding. In particular, prefetching of buckets could be used without virtual processors if the prefetching is non-binding. Nevertheless, the technique of simulating virtual processors is still applicable to these machines, and in fact, would be necessary on other machines that use a binding prefetch. For example, *E-registers* [175] on the Cray T3E could be used for simulating virtual processors with vector-like gather and scatter operations to buckets.

4.3.2 Reducing contention

We now consider how to avoid memory bank contention when mapping memory locations in the PRAM algorithm to memory modules in the (d, x)-BSP. In particular, contention can occur when permuting keys and when accessing buckets.

For sequential accesses to keys, it is practical to use a regular distribution of keys to memory banks, such as a cyclic distribution, which will naturally avoid any (significant) module map contention. However, regular distributions introduce the possibility of memory bank delays caused by patterns in the order messages are injected into the network. These delays could be reduced by randomizing the injection order of each processor. In practice, however, we have not found this to be necessary. Loop raking tends to break up common patterns that would cause the worst case permutations, and our implementation performs well on random inputs and on many regular inputs, such as a pre-sorted array.

Accesses to buckets pose a somewhat different, and more challenging problem than permutation of keys, because the number of accesses to each bucket depends on the distribution of values in the input keys. Since virtual processors are typically emulated on physical processors in a round-robin fashion, the key issue becomes the potential for bank conflicts among different virtual processors.

As discussed in Section 4.2, the buckets logically form a two-dimensional array in which the two indices identify the set of buckets for a virtual processor and the bucket within a set. One natural way to lay out the buckets would be to assign the buckets for each virtual processor to contiguous memory locations. This simplifies bucket addressing but can cause data-dependent bank conflicts on machines that cyclically distribute memory locations to memory banks. For example, with a radix of $2^8 = 256$, 256 virtual processors, and 256 memory banks, a virtual processor with a value of i will access memory bank i (assuming the buckets start at bank 0).

If several virtual processors handle keys with the same value, they will access the same memory bank and introduce a delay. In fact, the worst case is common, since some applications sort sets of keys in which all the keys have an identical digit. (A subtle example occurs when radix sorting an array of floating-point numbers in a small range. Even if the values are distinct, many of them will have the same exponent.)

This problem with cyclic distributions of bucket sets can be avoided by starting each set of buckets at a random offset. However, there is a better solution that both simplifies that address calculations and improves performance on the CRAY C90. Instead of using a randomized distribution, we distribute the buckets such that the buckets used by each virtual processor are in a separate memory bank (or a small set of memory banks). For example, virtual processor 0 might access only bank 0, whereas virtual processor 1 would access only banks 1.

Figure 4.1 quantifies the sensitivity to the distribution of key values for various choices of the module mapping. The graph indicates the time per key for generating a histogram of a single digit as the *key entropy* [194] is varied. In each graph, the key distribution consists of uniformly random numbers at the left and a single repeated value on the right. The naive module mapping is an order of magnitude slower for some key distributions. The worst-case histogram time is slightly above twice the bank delay, where the delay d is 14 clock periods on the CRAY J90 and 6 clock periods on the CRAY C90. Note that, for random key distributions (when the entropy per bit is 1.0), the CRAY C90 is more sensitive to network conflicts at the processor and thus benefits from the optimal mapping in which messages are injected in perfect cyclic order. However, there is no benefit to the conflict-free module mapping on the CRAY J90 for random keys because the network ports have sufficient buffering to avoid section conflicts.

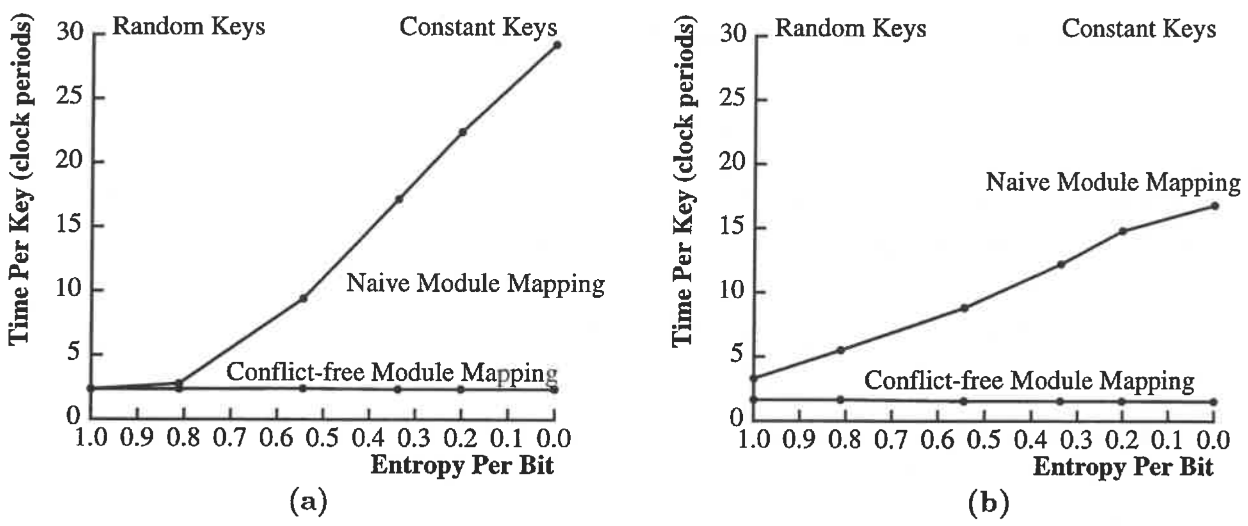


Figure 4.1: Histogram time for two different module mappings on a single processor of the (a) CRAY J90 and (b) CRAY C90.

4.4 Analysis of Cray implementation

This section presents an analysis of our sorting implementation for the CRAY C90. The analysis includes constant factors, which are critical for predicting the running times of sorting algorithms [148]. We first develop equations that predict the running time in terms of various parameters. These equations allow us to calculate the optimal values of free parameters, such as the radix and vector length, and allow us to predict the running time on any number of processors. The equations also allow us to factor out parameters specific to our implementation and to provide a degree of machine independence to our analysis. This section also describes the space used by the implementation and presents performance measurements, including a comparison with the CRAY library implementation and performance measurements on multiple processors of the CRAY C90. Except where noted, all timings were taken on random 64-bit keys. The timings in this section are based on Cray Assembly Language (CAL) [58] code generated from our own macro assembler.

We first consider the asymptotic running time for sorting b-bit keys with a radix size of r. The serial running time is

$$T = O\left(\frac{b}{r}(n + 2^r)\right) .$$

The running time on an EREW PRAM (with unit-time scans) is:

$$\begin{aligned} T &= O\left(\frac{b}{r}\left(\frac{n}{p} + \frac{p \cdot 2^r}{p}\right)\right) \\ &= O\left(\frac{b}{r}\left(\frac{n}{p} + 2^r\right)\right) \end{aligned}$$

The optimal value of r is $O(\lg(n/p))$, making the running time

$$O\left(\frac{b}{\lg(n/p)}\left(\frac{n}{p}\right)\right)$$

After noting the constant factors in our implementation, we describe how to choose the optimal radix, taking into account latency and constant factors.

4.4.1 Constant factors in the implementation

This section quantifies the costs for the three phases of the counting sort algorithm, filling in implementation details as necessary. All constants are reported in units of CRAY C90 clock periods (4.167 nsec). (Constants for the CRAY Y-MP are reported in [207].)

Histogram keys: The first step to building the histogram consists of clearing the buckets. Because each physical processor simulates v virtual processors, there are $2^r \cdot v \cdot p$ buckets, and we expect a speedup of p, yielding the equation:

$$T_{\text{Clear-Buckets}} = 0.6 \cdot 2^r \cdot v \cdot \frac{p}{p} = 0.6 \cdot 2^r \cdot v$$

To produce counts of the digits, each digit is converted to a bucket index, and the counts are gathered from the buckets, incremented, and scattered back to the buckets. The execution time for the histogram is:

$$T_{\text{Histogram-Keys}} = 1.3 \cdot \frac{n}{p}$$

Scan buckets: As discussed in Section 4.2, the bucket scan in the parallel algorithm is equivalent to executing a scan on the bucket matrix in column major order. In our implementation, the buckets are already arranged in column-major order, so we can simply use the plus-scan algorithm described in Section 3.3.3. The execution time is:

$$T_{\text{Scan-Buckets}} = 1.7 \cdot 2^r \cdot v \cdot \frac{p}{p} = 1.7 \cdot 2^r \cdot v$$

Rank and permute keys: This phase is similar to HISTOGRAM-KEYS, except that the keys are also loaded and permuted (scattered) using the offsets gathered from the buckets. Assuming a non-pathological permutation, the time for this step is approximately:

$$T_{\text{Rank-And-Permute}} = 2.9 \cdot \frac{n}{p}$$

The total sorting time can be characterized by two main parameters of the implementation, the time per bucket, T_{bucket}, and the time per key, T_{key} (expressed in clock periods per element per processor):

$$\begin{aligned} T_{\text{bucket}} &= T_{\text{Clear-Buckets}} + T_{\text{Scan-Buckets}} \\ &= 2.3 \\ T_{\text{key}} &= T_{\text{Histogram-Keys}} + T_{\text{Rank-And-Permute}} \\ &= 4.2 \end{aligned}$$

Since there are $v \cdot 2^r$ buckets per processor and n/p keys per processor, the time for one call to $T_{\text{Counting-Sort}}$ is:

$$T_{\text{Counting-Sort}} = v \cdot 2^r \cdot T_{\text{bucket}} + \frac{n}{p} \cdot T_{\text{key}} \tag{4.1}$$

Sorting b-bit keys requires $\lceil b/r \rceil$ passes of counting sort:

$$T_{\text{Radix-Sort}} = \lceil \frac{b}{r} \rceil \left(v \cdot 2^r \cdot T_{\text{bucket}} + \frac{n}{p} \cdot T_{\text{key}} \right) \tag{4.2}$$

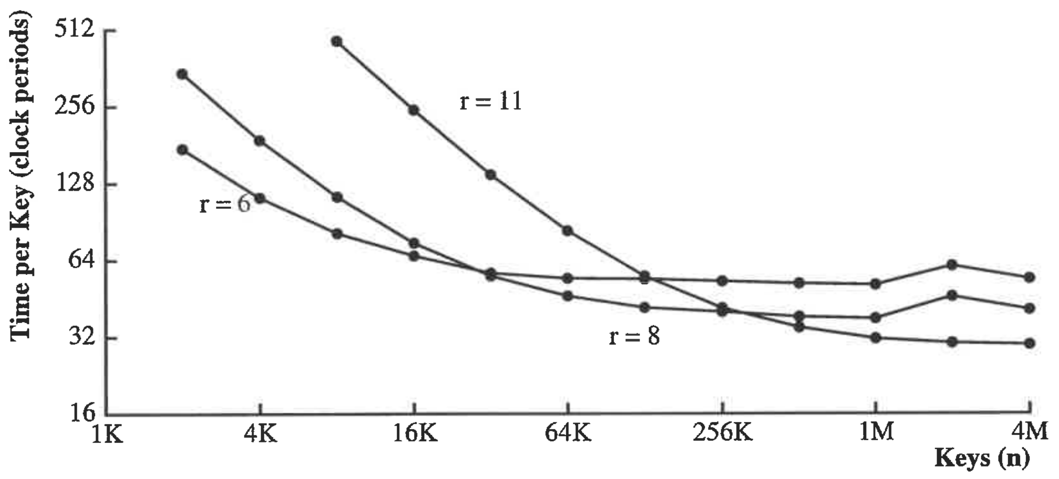


Figure 4.2: Time per key for a 64-bit sort on one processor of the CRAY C90 as the number of bits per pass (r) is varied. As the number of keys is increased, the optimal value for r increases.

4.4.2 Choosing the radix

As Figure 4.2 indicates, the optimal value for the digit size r increases with the number of elements per processor. For a given problem size, choosing r below the optimal value will cause too much work on keys, whereas choosing r above the optimal value will cause too much work on buckets, as illustrated in Figure 4.3. We can determine the value for r that minimizes the total time by differentiating Equation 4.2 with respect to r and setting the derivative equal to zero. The value for r that we obtain satisfies:

$$\begin{aligned} r &= \lg\left(\frac{n}{p} \cdot \frac{T_{\text{key}}}{v \cdot T_{\text{bucket}}}\right) - \lg\left(r \ln 2 - 1\right) && (4.3) \\ &\approx \lg\left(\frac{n}{p}\right) - \lg(v) - 1 && (4.4) \end{aligned}$$

where we approximate T_{key} with $2 \cdot T_{\text{bucket}}$ and approximate the second term of Equation 4.3 with the constant 2, which is derived from setting r to a moderate value of 7.

For $n = 32K$, as in Figure 4.3, the optimal value for r predicted by Equation 4.4 is $r \approx 8$, which, in fact, minimizes the total time.

When we substitute the approximation for r back into Equation 4.2, we obtain the following approximation for the total sort time:

$$T_{\text{Radix-Sort}} \approx \frac{b \cdot (n/p)}{\lg(n/p) - \lg(v) - 1}\left(\frac{T_{\text{bucket}}}{2} + T_{\text{key}}\right) \quad (4.5)$$

Figure 4.4 compares this equation with the actual running time of our implementation of radix sort in which the best experimental values for r are used. As the figure indicates, this equation accurately predicts the running time.

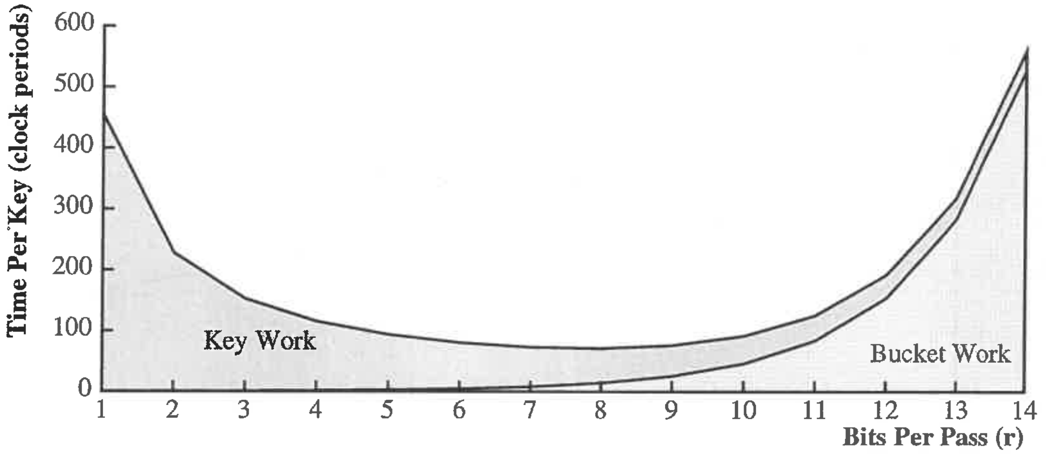


Figure 4.3: Breakdown of the total running time of radix sort into the time spent on buckets and the time spent on keys. Times are derived from two terms of Equation 4.5 for a one-processor 64-bit sort of 32K elements. As r is increased, the work per bucket increases and the work per element decreases. For the parameters chosen, the optimal value for r is 8, as indicated by the top curve representing the total time.

Note that the time per key *decreases* as the number of keys increases, because we can increase the radix. In contrast, the time per key for comparison-based sorts is typically proportional to $\lg n$, and thus increases for larger sorting problems. Furthermore, the time for radix sort is proportional to the number of passes of counting sort, i.e., the number of digits. Thus, we can sort 32-bit keys approximately twice as fast as 64-bit keys, and 20-bit keys approximately three times as fast. In contrast, comparison-based sorts typically take the same amount of time for shorter keys. Shorter keys are important because some applications sort indices or pointers that are significantly shorter than 64 bits.

4.4.3 Latency tradeoffs

In the previous analysis, we assumed that latency was fully hidden with the use a large number of virtual processors. In particular, we used the maximum allowable vector length (128 for the CRAY C90) and accounted for the relatively small cost of latency by including the cost in the measured constants, rather than modeling latency as separate component of the running time. This section analyzes tradeoffs between latency and work using the (d, x)-BSP framework to model the cost of latency.

One superstep corresponds to one step of the algorithm for each virtual processor (e.g., incrementing a histogram bucket on each virtual processor) and is implemented with a series of vector operations (e.g., gather, add, scatter). The overhead for a superstep is amortized over the virtual processors simulated in the algorithm. Note that the following analysis models the

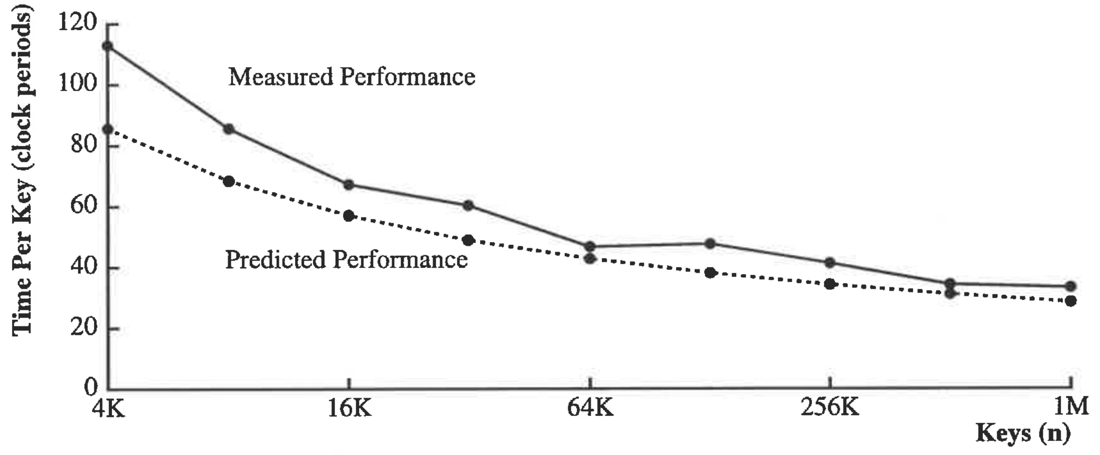


Figure 4.4: Predicted and measured performance of radix sorting 64-bit keys on one processor of the CRAY C90. Measured performance uses the empirically determined optimal values for r. The predicted performance was calculated using Equation 4.5.

overhead using the memory latency parameter (ℓ), rather than the periodicity parameter (L), because the histogram and ranking operations performed on each processor are independent. Synchronization is necessary only between phases of the algorithm (e.g., after completing the histogram and before starting the scan), and is not required between supersteps. (On some machines, a *memory barrier* might be required between supersteps, but global synchronization is not necessarily required.) A refinement of Equation 4.1 includes an additional term for each of the $n/(v \cdot p)$ supersteps:

$$T_{\text{Counting-Sort}} = v \cdot 2^r \cdot T_{\text{bucket}} + \frac{n}{p} \cdot T_{\text{key}} + \frac{n \cdot l}{v \cdot p} \tag{4.6}$$

By using fewer virtual processors (and thus fewer buckets), the amount of work can be reduced. This decreases the time for clearing and scanning the buckets, T_{bucket}, at the expense of increasing the cost per element for performing the histogram, T_{key}, due the increased cost of latency. Figure 4.5 shows the effect of varying the vector length. Figure 4.6(a) shows the optimal vector length, and Figure 4.6(b) shows how the optimal radix is affected by the selection of the vector length. In practice, it works well to use Equation 4.4 for r, but with a minimum value of $r = 4$, and to set the vector length to $v = \min(2.0 \cdot \sqrt{n/p}, V_{\max})$, where $V_{\max} = 128$ on the CRAY C90. (We always use a power-of-two vector length in order to compute bucket indices with shift instructions rather than multiply instructions.)

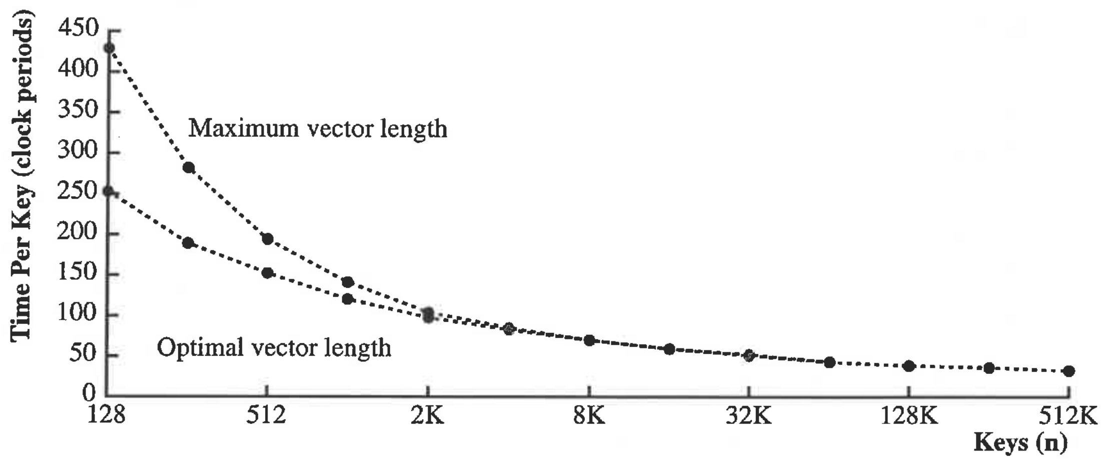


Figure 4.5: Predicted effect of varying the vector length for a 64-bit sort on the CRAY C90. For small problem sizes, varying the vector length leads to significant improvements over using the maximum vector-register length of 128.

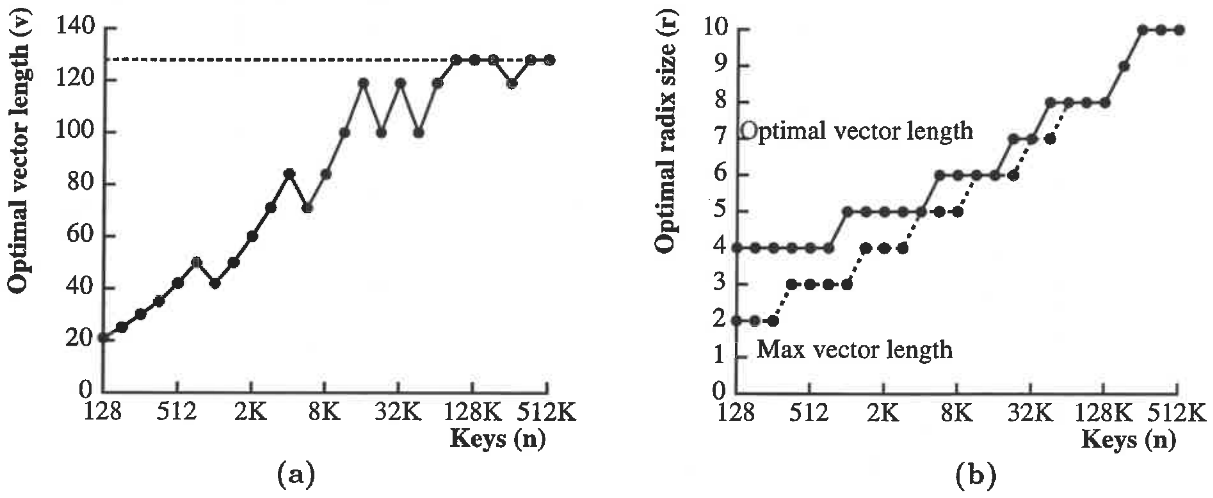


Figure 4.6: Optimal vector length and radix on the CRAY C90 based on Equation 4.6. (a) shows the optimal vector length as a function of the number of keys per processor. For large problems, the optimal vector length is 128. (b) shows that the optimal radix is affected by whether the vector length is fixed at the maximum vector-register length or varied with the problem size.

4.4.4 Space

An important concern for designing a practical sorting routine is to minimize the space required by the implementation, as well as the running time. To sort an array of size n, our sort requires an array of size n for the destination of the permute and an array of size $v \cdot p \cdot 2^r$ for the buckets. Since the optimal value of r is approximately $\lg(n/p) - 7$, the amount of memory for the buckets will be approximately $v \cdot p \cdot 2^{lg(n/p)-7}$, which is simply $\frac{n}{2}$. Thus, the total amount of memory used besides the source is approximately $1.5n$. Memory could be saved at the expense of increased execution time by using a lower radix or by reducing the vector length.

4.4.5 Sorting interface

In practice, a sorting routine that can carry additional data with keys or can sort records is more useful than one that can only sort keys. A clean way to provide a more general sorting interface is to implement a routine that returns a permutation that can be used to sort data. Such an interface is useful for implementing a multipass routine for sorting records, since the permutations returned from sorting on each field of the record may be composed. It also makes sorting large data items with small keys more efficient, since only the keys are manipulated internally.

There are two common variants of sorting routines that return permutations: RANK and ORDERS. RANK takes a n-element input array *Key* and returns an n-element permutation *Perm*, such that *Perm*[i] is the rank of *Key*[i]. This permutation can be used for "scattering" the keys into sorted order. ORDERS returns the inverse permutation, an n-element permutation *Perm*, such that *Perm*[i] is the index of the ith smallest key. This permutation can be used for "gathering" the keys into sorted order. RANK is standard in HPF and ORDERS is the standard library routine callable from Fortran on the CRAY [57].

We implemented both ORDERS and RANK routines and will describe ORDERS (RANK is similar). To transform our sorting implementation into an ORDERS routine, a *current permutation* is stored and is permuted on each pass rather than permuting the keys. We start by initializing the current permutation *Perm* to the identity permutation, such that *Perm*[i] $= i$. We modify the histogram phase to gather the source data using the current permutation, rather than loading it directly, and we modify the permutation phase to permute the *Perm* array rather than the keys.

4.4.6 Running time

This section discusses absolute performance and scalability, by comparing our results to timings of the NAS Parallel Benchmarks and the library sorting routine from Cray Research.

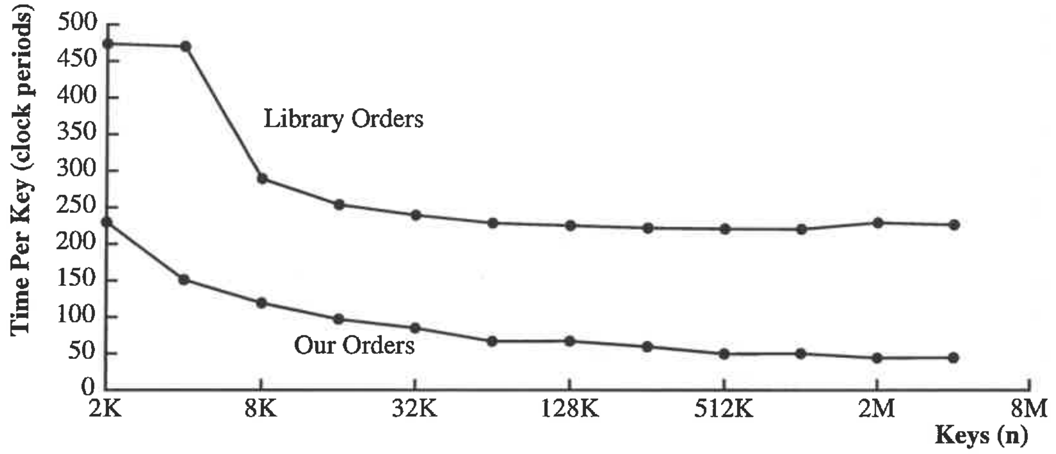


Figure 4.7: Comparison of CRAY library ORDERS and our implementation of ORDERS on 64-bit random keys using one processor of the CRAY C90. ORDERS is a version of sort that returns the positions of the sorted keys rather than permuting the keys. Our implementation of SORT is about 10 percent faster than our implementation of ORDERS. (The discontinuity in the library version is due to a change in radix from 2^8 to 2^{16}.)

We begin by comparing our implementation of ORDERS to the CRAY optimized library routine, which is also a radix sort, to demonstrate that the results we obtain are due to an algorithmic improvement, rather than tighter code. The library routine uses either $r = 8$ or $r = 16$ as determined by the caller, and judging from the temporary memory requirements, it does not use multiple sets of buckets. Figure 4.7 graphs the performance of our implementation and the library implementation. The timings use random keys. (Interesting, the CRAY library sort is approximately 1.8 times slower when sorting all zeros.) As the figure indicates, our implementation is 2 to 5 times faster for sorting more than 2K items.

Equations 4.4 and 4.5 predict the theoretical speedup on p processors. Note that Equation 4.5 expresses the total time as a function of the number of elements per processor; the radix is chosen based on n/p and the total time is proportional to n/p. These relationships imply that speedup will be linear when increasing the number of processors, if n/p is held constant, that is, if the problem size grows with the number of processors. However, if the problem size, n, is fixed as we increase the number of processors, the speedup is not linear, because the optimal radix is lower. For example, with $n = 256K$, the optimal value for r on 1 processor is $\lg(n/1) - 7 = 11$, whereas the optimal value on 8 processors is $\lg(n/8) - 7 = 8$. Thus, with 1 processor, we need 6 passes of counting sort to sort 64-bit keys, but with 8 processors, we need 8 passes, yielding a maximum speedup of 6 on 8 processors. Using 16 processors on the NAS Integer Sort benchmark, we achieve a speedup of approximately 13 on 16 processors. For this benchmark, the same problem size and radix is used for all processor counts, which

Machine	Processors	IS (C90 equiv.)
Cray T3E	256	16.15
NEC SX-4	16	14.42
Cray C90	16	13.18
Cray T3D	1024	10.59
Cray T90	8	9.16
IBM SP2/P2SC	64	7.38
Fujitsu VPP 300	16	4.74
Cray J90	16	2.63
SGI Origin 2000	32	2.24
Convex SPP1000	8	0.30

Table 4.1: Performance on the NAS Class B Integer Sort (IS) benchmark [16, 172, 173]. The performance is expressed as a throughput measure normalized to the performance of a single-processor CRAY C90.

indicates that the speedup is limited primarily by saturating the memory system. In general, linear speedup is not achieved due to two effects: saturating the memory system causes a small degradation in effective memory bandwidth, and using a separate set of buckets for each virtual processor forces us to use a slightly lower radix.

Table 4.1 presents timings on the NAS Integer Sort benchmark (Class B, version 1.0) for various machines. The benchmark measures the time to rank 10 sets of 32 million 21-bit integer keys. Our implementation (used on the CRAY J90, CRAY C90, and CRAY T90) is a Fortran implementation of the algorithm described in this chapter. (Other timings in this chapter refer to the assembly language implementation.) In contrast to several of the microprocessor-based implementations, our implementation is not specialized to the specific key distribution used in the benchmark. Accurate pricing data is more difficult to obtain, but pricing estimates reported by NAS [172, 173] suggest that the CRAY J90 is the leader in price-performance, closely followed by the NEC SX-4.

4.5 Discussion

The chapter described an efficient mapping of an EREW PRAM algorithm to (d, x)-BSP. A straightforward use of virtual processors gives excellent latency tolerance, and careful selection of module mapping avoids memory bank contention. The implementation has reasonable memory usage, supports two types of ranking primitives in addition to simple sorting, produces stable orderings of equal keys, and has predictable running time. The implementation is fast over wide range of problem and machine parameters, both in comparison to previous vector sorting implementations and sorting implementations on other architectures. These results show that the combination of high-bandwidth pipelined memory and effective mapping techniques leads to high performance on a general-purpose sorting algorithm.

Chapter 5

Merging

This chapter studies the problem of merging two sorted arrays. We consider variants of merging algorithms designed for the CREW and EREW PRAM models, and compare the performance of work-efficient, irregular PRAM algorithms to simpler, less efficient algorithms that have been previously used on vector and distributed memory machines. Cost models are presented for accurately predicting running times and for exploring tradeoffs between latency hiding and work. The impact of contention is studied by experimenting with a set of test cases. Techniques for hiding latency lead to significant performance improvements over previous work on vector multiprocessors. For large lists, an implementation for a single CRAY C90 processor performs approximately 8 times faster than a serial merge and nearly a factor of $\lg n$ faster than bitonic merge. On a 16-processor CRAY C90, absolute performance is approximately 500 million elements per second, which to our knowledge is the fastest implementation ever reported.

The chapter is organized as follows. Section 5.1 reviews previous work on merging. Section 5.2 describes several practical PRAM merging algorithms. Section 5.3 describes techniques for hiding latency and analyzes trade-offs between latency and work. Section 5.4 quantifies the effect of concurrent reads using the $(d, \mathbf{x})$-BSP and describes several approaches to reducing contention, and Section 5.5 discusses our results.

5.1 Background and previous work

This section presents an overview of previous work in parallel merging. (See Table 5.1). We consider both theoretical and practical issues, with the goal of identifying algorithms that are both work-efficient and practical. Parallel algorithms for merging can be classified into two categories: merging networks and PRAM algorithms. Merging networks, such as Batcher's bitonic merging network [19], consist of two-element comparators that output the minimum and maximum of their two inputs. One advantage to algorithms based on simulating a merging

Algorithm	Model	PRAM Running Time
serial	RAM	$O(n)$
bitonic network	EREW	$O((n/p)\lg n)$
bitonic hybrid	EREW	$O((n/p)(\lg p+1))$
CR-split	CREW	$O(n/p+\lg n)$
CR-block-split	CREW	$O(n/p+\lg n)$
ER-block-split	EREW	$O(n/p+\lg n)$

Table 5.1: Summary of merging algorithms discussed in this chapter.

network is that the data access patterns are *oblivious* (also called *non-contingent*), that is, the control flow does not depend on the input values. Oblivious algorithms tend to have simpler implementations, lower constants, and greater predictability. However, these benefits come at the expense of additional work compared to a work-efficient algorithm. For example, simulating an n-input bitonic merging network, which has a size of $O(n \lg n)$ and a depth of $O(\lg n)$, requires an additional factor of $\lg n$ work.

The inefficiencies in using merging networks motivate the use of work-efficient merging algorithms designed for the PRAM. There are a number of similar PRAM merging algorithms that work by partitioning the merging problem into independent merging subproblems. This partitioning requires finding a number of *splitter* elements that split the input values into ranges containing approximately the same number of values. The algorithms differ in how they select the number of subproblems, how the ranges are identified, and how balanced the subproblem sizes are.

Valiant [200] presented an efficient parallel merge that runs in $O(n/p+\lg\lg n)$ on a CREW PRAM [112]. Valiant's algorithm is complicated to implement because of its use of recursive merging. Handling the recursive calls of uneven sizes requires dynamic processor allocation (as described by Shiloach and Vishkin [178]), which would lead to large constant factors in an implementation. For a practical implementation, we would like an algorithm that performs only a single complete pass over the input lists, provides an optimal load balance for an arbitrary number of processors (with a small amount of bookkeeping), uses a minimal amount of temporary memory (beyond having a separate output list), and has low memory contention. Several studies have presented practical merging algorithms that are close to this ideal. For example, Varman et al. [202] and Francis and Mathieson [78] implemented practical mergesort algorithms for Sequent bus-based, shared-memory multiprocessors [138]. The implementations achieve near-linear speedups over the sequential algorithm, but both studies use algorithms with concurrent reads.

Despite the success in developing practical implementations for small shared memory machines, several implementations of merging for massively parallel processors and vector processors have not used work-efficient algorithms. For example, studies of sorting algorithms for the CM-2 [34] and the CMU/Intel iWarp [188] both made use of bitonic merge, an algorithm that performs a logarithmic factor more work than optimal. Previous implementations of vectorized merging include an implementation of bitonic sort on the Cray X-MP and Fujitsu VP-200 by Overill [155] and an implementation of Batcher's merge-exchange sort [19] on Cray and Amdahl machines by Rönsch and Strauss [170].

Interestingly, practical optimal merging algorithms for shared memory models have not been applied to vector computers. In fact, Torii et al. [196] proposed special-purpose hardware extensions for the Hitachi for vectorized merging (and searching). One contribution of this chapter is to describe how parallel merging can be implemented by applying *general* techniques for mapping PRAM algorithms onto pipelined processors.

5.2 Practical PRAM merging algorithms

We describe algorithms for merging two ordered arrays, A and B, of size n_a and n_b into an ordered output array, C, of size $n = n_a + n_b$. Ordering is defined by a comparison function denoted by the symbol $\leq$. Throughout the following discussion, we assume n is an exact multiple of p.

Bitonic network merging: A bitonic merge [19, 56] uses a comparison network that outputs a sorted sequence given a bitonic sequence, a sequence that monotonically increases and then monotonically decreases (or can be circularly shifted to become so). The bitonic merging network (see Figure 5.1) has n levels of $\lg n$ comparators, and thus the running time on a PRAM for simulating the network is $O(n/p \cdot \lg n)$.

Hybrid bitonic merging: In Figure 5.1, note that the bitonic merging has a divide-and-conquer structure. This structure is used in hybrid bitonic merge in order to replace a simulation of the last $\lg(n/p)$ levels of the bitonic merging network with p independent sequential merges of n/p elements. For large problem sizes, the resulting algorithm is a significant improvement over straight bitonic merging, but still performs a factor of $\lg p$ more work than a work-efficient algorithm. The running time on a PRAM is $O(n/p \cdot (\lg p + 1))$.

CR-split: First, we consider a simple yet efficient parallel merging algorithm described by Francis and Mathieson [78]. The running time on a CREW PRAM is $O(n/p + \lg n)$, and from a practical standpoint, the algorithm has very small constant factors. The algorithm finds optimal

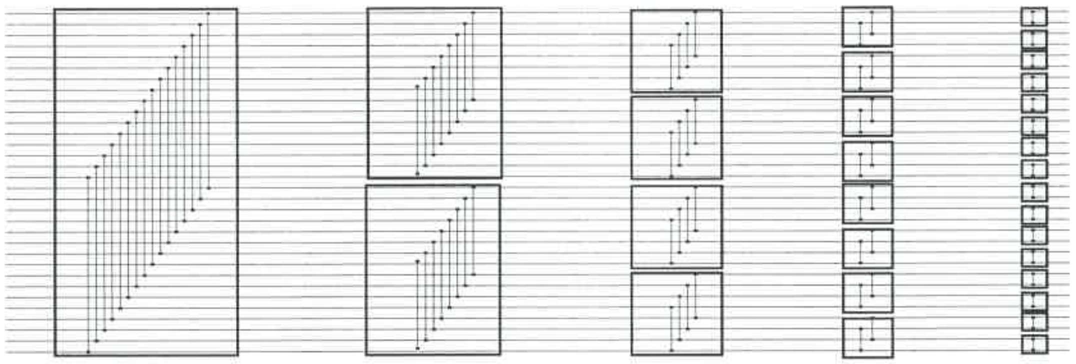


Figure 5.1: A bitonic merging network. The merging network accepts $n = 32$ elements at the left, comprising an ascending sequence and a descending sequence, each of size $n/2$. Each of the $\lg(n)$ levels of the merge has $n/2$ comparison elements.

"splitters" that divide the merging problem into p merging problems that can be solved using a conventional serial merging algorithm. We refer to this algorithm as "CR-split". For large lists, the expected speedup is exactly p because the total number of elements assigned to each subproblem is optimally load-balanced.

Finding splitters requires identifying where the kth smallest element in the result C appears in A and B, for each $k = (n/p) \cdot [0, 1, 2, \ldots, p-1]$. These elements can be located using *dual binary search*, a search that probes both A and B simultaneously, and on each iteration halves the candidate range of one of the lists [78]. Given the set of splitters, the merging task is embarrassingly parallel on a PRAM, and there is no "cleanup phase" needed for concatenating results, as with more complicated merging algorithms.

CR-block-split and ER-block-split: Shiloach and Vishkin [178] and Varman et al. [202] described a similar, slightly more complicated algorithm with some significant advantages for reducing memory contention. We refer to this algorithm as "CR-block-split", because it divides the input lists into blocks before finding splitters.

The algorithm performs the following steps:

1. *find block pairs*: Conceptually divide the input lists into blocks of approximately n/p elements. Determine the blocks where partitions are located by merging *representatives* of each block; representatives are the smallest element of each block of A and the largest element of each block of B. The result is a list of p pairs of block numbers that indicate which blocks contain the desired partition indices.

2. *find splitters*: For each pair of overlapping blocks (one block from A, and the other from B) use binary search to find the *crossover point*. The crossover point is the point at which

the values from A begin to exceed the values from the corresponding block of B in reverse order. The crossover points can be used to find the splitter elements in the input lists.

3. *parallel merge*: Merge pairs of lists starting at the splitter elements. The merging step performs only exclusive reads and writes, is embarrassingly parallel, and is optimally load-balanced.

If the splitter-finding is performed serially, the entire merging algorithm runs in $O(p + p\lg\frac{n}{p} + \frac{n}{p})$, which is optimal for $p = \sqrt{\frac{n}{\lg n}}$. If the splitter-finding is parallelized using bitonic merge, the entire merging algorithm runs in $O(n/p + \lg n)$, which is optimal for $p = \frac{n}{\lg n}$.

This algorithm has two main advantages over the simpler CR-split PRAM algorithm. First, contention is reduced because the binary searches do not all start at the same index (except in pathological corner cases). Second, the pairing of block provides distribution information that can be used to estimate the contention in the splitter finding and parallel merge phases. This distribution information can be used in an EREW version that broadcasts any input blocks read by multiple processors [202]. We refer to this variant as "ER-block-split".

5.3 Modeling latency hiding in merging algorithms

This section describes latency hiding for merging algorithms using virtual processors. We analyze several merging algorithms, including two variants of bitonic merge, and two variants of a work-efficient PRAM merging algorithm. For each algorithm, we describe tradeoffs between latency and work. We then compare the predicted and measured performance of the algorithms on the CRAY C90.

In the equations below, the key parameters are: the problem size (n), the number of processors (p), the number of virtual PRAM processors simulated per physical processor (v), and the memory latency (ℓ). Note that v is essentially a "slackness" parameter. Also note that memory latency parameter ℓ includes only the memory latency, in contrast to the BSP parameter L which also includes the time for global synchronization. The calculations for the CRAY C90 use a value of 50.0 for the latency ℓ. (This is somewhat higher than the hardware memory latency because it includes software overhead.)

Constant factors are given for the CRAY C90 implementations. The implementations are conceptually straightforward, but in both the serial and parallel implementations of merging, conditional computation is simplified by adding guard elements at the end of each of the input lists.

The equations for parallel algorithms model latency by expressing running times as a product of two terms: the time per virtual processor and the slowdown of simulating v virtual processors per physical processor. For example, an embarrassingly parallel PRAM algorithm with a running time of $O(n/p)$ will have a running time of:

$$O((n/(v \cdot p)) \cdot (\ell \oplus v))$$

where $\oplus$ denotes either a maximum or addition operator. (We use addition because it produces more accurate predictions for our CRAY implementation.) Note that the running time is simply $O(n/p)$ for large values of v relative to ℓ. For small values of v, the slowdown term will cause an increase in the running time.

Serial: Serial merging serves as a baseline for comparison with other algorithms. It is quite fast in practice because it requires only one pass over the input data, performing a single comparison for each output element. However, in a straightforward implementation with no overlap of memory accesses, each access must incur the cost of the memory latency ℓ, yielding a running time of $O(n \cdot \ell)$. (If the memory accesses can be fully overlapped, for example, with prefetching or long cache lines, then the running time is simply $O(n)$.) Our serial implementation was optimized using manual loop unrolling and is approximately twice as fast as a naive implementation. The running time is:

$$T(n) = c_{\text{serial}} \cdot \ell \cdot n$$

where c_{serial} is approximately 1.0 on the CRAY C90.

Bitonic network merging: If we assign one virtual processor per row of comparators in the bitonic merging network (Figure 5.1), the asymptotic running time is the product of the time per virtual processor ($O(\lg n)$) and the slowdown for simulating v virtual processors:

$$T(n,p) = O(\lg n \cdot (\ell + v))$$

The number of virtual processors v is simply n/p, yielding the following equation for the running time:

$$T(n,p) = c_{\text{bitonic}} \cdot \lg n \cdot (\ell + n/p)$$

where c_{bitonic} is approximately 3.5 on the CRAY C90.

For all but the smallest problems, latency is trivial to hide for bitonic merge. For small problem sizes, the optimal number of virtual processors is simply n.

Hybrid bitonic merge: In the hybrid bitonic merge algorithm, the network simulation phase has $O(\lg(p \cdot v))$ steps with $O(n/p)$ slackness and the independent merging phase has $O(n/(p \cdot v))$ steps with $O(v)$ slackness. Thus, the asymptotic running time is

$$O(\lg(p \cdot v) \cdot (\ell + n/p) + (n/(p \cdot v)) \cdot (\ell + v))$$

In more detail, the running time is:

$$T(n,p,v) = c_{\text{bitonic}} \cdot (\lg(v \cdot p) - 1) \cdot (\ell + n/p) + c_{\text{merge}} \cdot n/(v \cdot p) \cdot (\ell + v)$$

where c_{merge} is approximately 4.5 on the CRAY C90.

For small n (i.e., $n/p < \ell$), the optimal value of v is simply $O(n/p)$, which indicates that there is no benefit over the bitonic network merge algorithm. For larger n, the optimal value of v is $O(\ell)$, i.e., proportional to the memory latency. Since the optimal slackness is based on the latency, rather than the maximum slackness, an optimal implementation for a vector computer will not necessarily use the maximum vector register length, even for large n.

CR-split: The asymptotic time for the CR-split algorithm is the sum of the time for searching:

$$O\left(\lg(n/p) \cdot \max(\ell + v)\right)$$

and the time for independent parallel merges:

$$O\left(\frac{n}{p \cdot v} \cdot (\ell + v)\right)$$

Thus, the total running time is:

$$T(n,p,v) = (c_{\text{search}} \cdot \lg(n) + c_{\text{merge}} \cdot n/(p \cdot v)) \cdot (v + l)$$

where c_{search} is approximately 20.0 on the CRAY C90.

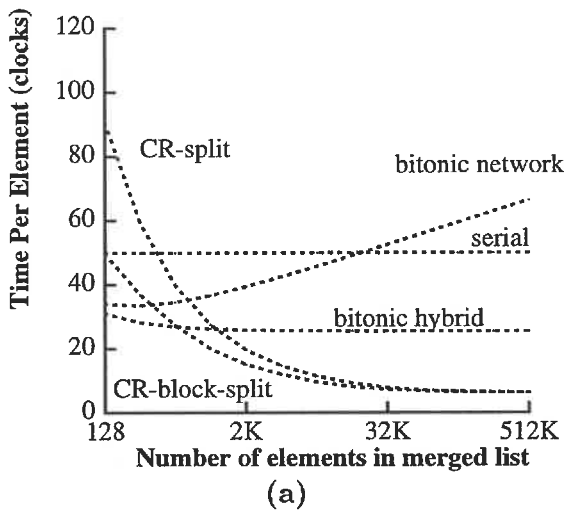
Small problem sizes present a tradeoff between the time for finding the splitters and the time for merging. The selection of an optimal v is explored shortly.

CR-block-split: The analysis for CR-block-split is similar to that of CR-split. The analysis is based on the CRAY implementation, which uses a serial implementation to find block pairs and find splitters. The merging time is the same as that of CR-split, and the time for serial finding block pairs and finding splitters is:

$$O\left(p \cdot v \cdot \ell \cdot (1 + \lg \frac{n}{p})\right)$$

The total running time is:

$$T(n,p,v) = p \cdot v \cdot \ell + (c_{\text{search}} \cdot \lg(n/(p \cdot v)) + c_{\text{merge}} \cdot n/(p \cdot v)) \cdot (v + \ell)$$



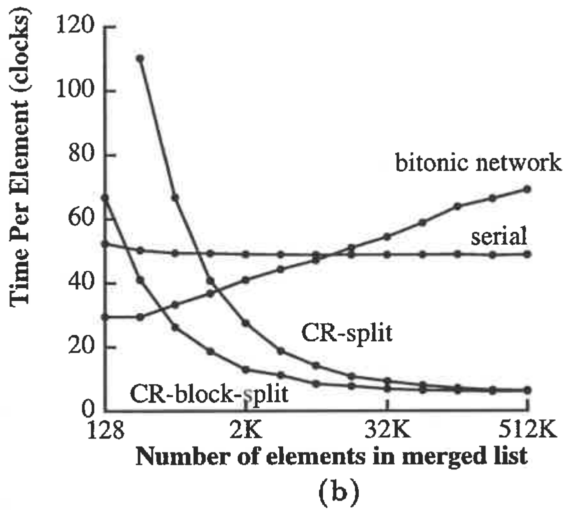
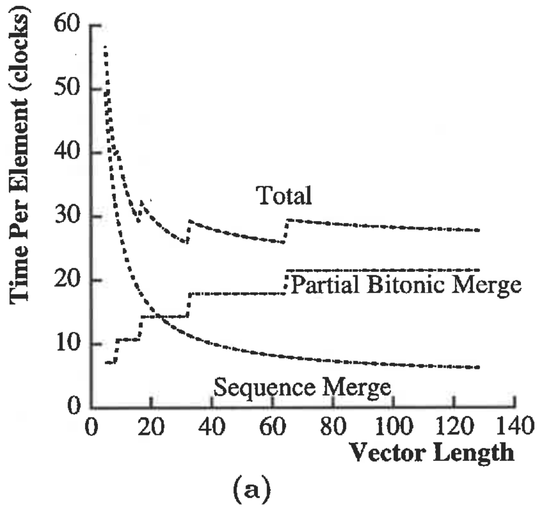


Figure 5.2: **(a)** Predicted and **(b)** measured merging performance on the CRAY C90.



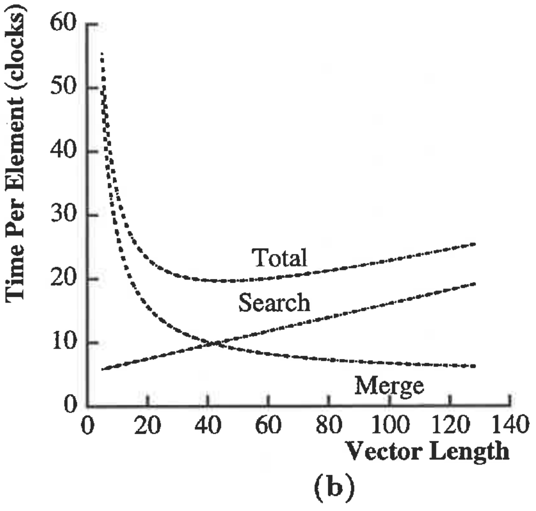


Figure 5.3: Predicted breakdown of running time as a function of vector lengths (with a fixed problem size, n=2048) for **(a)** hybrid bitonic merge, and **(b)** CR-split.

Analysis of algorithm parameters: The performance equations can be used to understand how performance is affected by various parameters, such as problem size, number of processors, and the memory latency. We can examine the running times (as shown in Figure 5.2), the effect of varying the vector length (as shown in Figure 5.3), the optimal vector length as a function of problem size (as shown in Figure 5.4(a)), and the sensitivity of the algorithms to memory latency (as shown in Figure 5.4(b)).

Figure 5.2 compares the performance of several merging algorithms on one vector processor using parameters for the CRAY C90. The input lists are generated randomly. (Other test cases are considered in Section 5.4.) At small problem sizes, there are discrepancies due to fixed overheads, but we get a clear picture of the relative performance of algorithms, particularly for large problem sizes. For large problem sizes, the PRAM algorithms are approximately eight times faster than the serial algorithm and over ten times faster than the bitonic network algorithm.

Figure 5.3 shows a breakdown of running time for a small problem size of n=2048 elements. Figure 5.3(a) shows that the running time is minimized for hybrid bitonic merge when the vector length is 32. (With a vector length of 64, the time is within one percent of optimal.) Figure 5.3(b) shows that the running time is minimized for the CR-split algorithm when the vector length is near 45.

Figure 5.4(a) shows that the optimal vector length for the PRAM algorithms gradually increases with the problem size. At small problem sizes, the cost of searching is significant and thus the optimal vector length is small. At larger problem sizes, the cost of merging dominates, and the optimal vector length is large in order to hide latency in the merging. For the bitonic merge, the amount of work is independent of the problem size (for problem sizes that are greater than the maximum vector length, 128), and thus the optimal vector length is always 128. For hybrid bitonic merge, there is a tradeoff between the time for the partial bitonic merge and the sequence merges. Note that the optimal vector length is 64 even for large problem sizes, because using a higher vector length would increase the amount of work in the partial bitonic merge.

Figure 5.4(b) shows that the serial algorithm is the most sensitive to memory latency, as expected. In contrast, bitonic merge is independent of the memory latency, (for all but the smallest problem sizes) because the amount of inherent parallelism grows linearly with the problem size. The PRAM algorithms and hybrid bitonic algorithms are somewhat sensitive to memory latency.

5.4 Modeling and reducing contention

In this section, we identify a number of sources of contention, outline several methods for reducing contention, and present experimental measurements of contention.

5.4.1 Sources of contention

Memory location contention: There are two potential sources of contention in the find-splitters and merge algorithm. One is memory contention when beginning the binary searches: each virtual processor begins searching at the same positions in the input lists, which leads to worst-case contention. (See Figure 5.5(a).) This contention can be reduced by using a binary search fat-tree algorithm for the QREW model (see Section 3.3.1). Alternatively, the starting

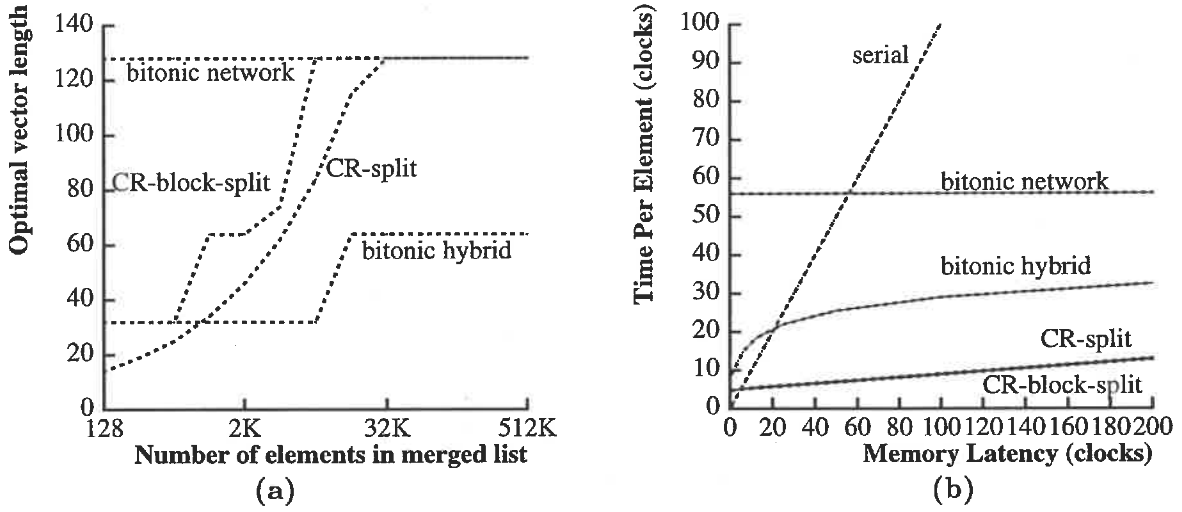


Figure 5.4: Parameter variations for various merging algorithms. **(a)** Optimal vector length as a function of problem size. **(b)** Time as a function of memory latency (for a fixed problem size, n=65536).

points can be skewed by picking a random starting point in one list and a corresponding starting point in the other list that makes the sum of the starting offsets $n/2$.

Empty sub-lists are a second potential source of contention. Techniques such as the binary search fat-tree reduce the contention early in the search and work well for many inputs, but can in some cases still lead to high contention towards the end of the search. For example, when all the elements of one input list are smaller than all the elements of the other list, the result of the partitioning will cause half the sublist pointers to point to the first element in one list and the other half to point to the last element in the other list. When merging in this case, there will be nearly worst-case contention on each step of the merge. In general, read contention is a problem whenever there are large blocks of the output that will be coming from a single input list. Possible solutions including treating empty input sub-lists as a special case, reloading elements from input lists only when pointers are advanced, and replicating guard elements on sub-lists used by multiple virtual processors.

Bank and network contention: For machines that support pseudo-random mappings of memory locations to banks, minimizing memory contention is sufficient for avoiding bank and network contention. However, for machines with regular mappings, such as the Cray vector machines, there are some pathological inputs that can cause bank and network contention, despite having low memory contention.

One source of bank and network contention is locality in adjacent virtual processors. If we distribute virtual processors to physical processors using the typical block distribution and also

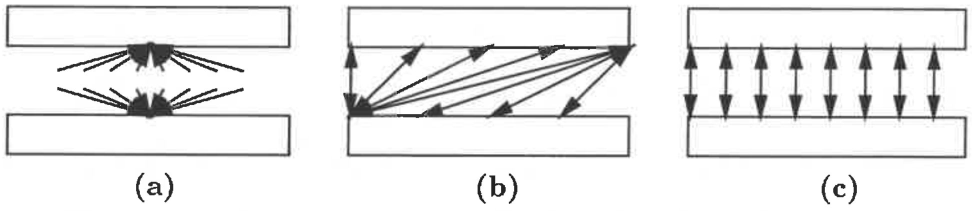


Figure 5.5: Sources of contention in merging. **(a)** depicts memory contention in the binary search, in which all searches beginning at middle element of the input lists. **(b)** and **(c)** depict two other cases that can cause contention. The arrows indicate positions where parallel merges will begin in the two input lists. In **(b)**, memory contention in merging is caused by when one list contains elements that are all larger elements of the other list. In **(c)**, regular input patterns cause regular strided accesses. If the stride is a power of two, bank contention could be high on an interleaved memory system.

allow a modest amount of contention, this will cause network contention because the requests to the same location will be made in sequence. This problem can be alleviated by using a random or cyclic assignment of virtual processors to physical processors. The motivation for using a cyclic distribution is that with processors running asynchronously, it is more important to break up regularity within a processor, than across processors. Reordering virtual processors is fairly inexpensive to implement, although in some cases it can increase the amount of indirect addressing.

A second source of contention is due to regular access with bad strides. To avoid regular accesses with large power-of-two strides, we can adjust the number of virtual processors. Loop raking (described in Section 3.1.3) is used for strided vector operations, for example, when accessing the output array C. A more subtle problem arises when regular patterns in the input data cause contention during the merging. (See Figure 5.5(c).) Although this is uncommon, it could be solved by simulating random hashing, or by inserting "bubbles" with random sizes between input sub-lists. Another possibility is to program directly for the (d, x)-BSP and explicitly assign memory locations to specific modules. The two source arrays for each virtual processor can be assigned to a module to get conflict-free merging with optimal injection. (A similar technique is used for sorting in Chapter 4.) This redistribution could be implemented by moving the elements to their destinations in random order, or by carefully orchestrating the ordering to avoid network conflicts, possibly by extending Bailey's conflict-free matrix transpose algorithm [14] to ragged arrays.

An alternative to modifying one of the CREW algorithms is to use the ER-block-split [202] algorithm. The algorithm broadcasts input blocks that are read by multiple (virtual) processors. Figure 5.7(b) shows the effect of replication in the ER-block-split algorithm for block cyclic test cases. For these experiments, each block of both input lists is copied to a temporary array.

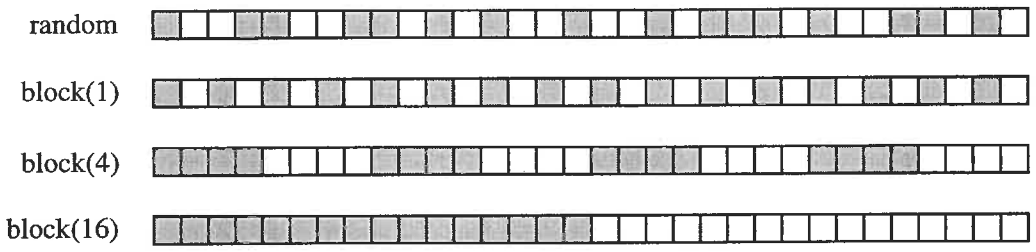
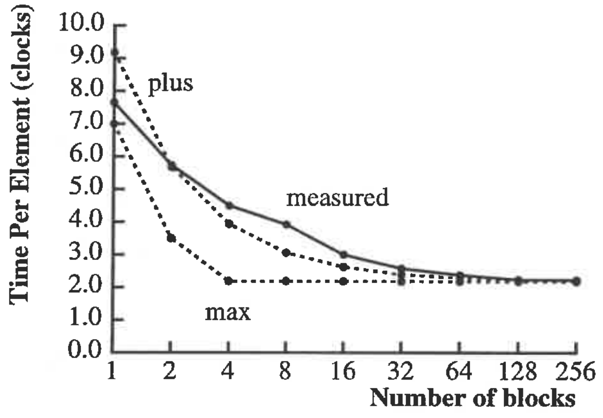


Figure 5.6: Test set for merging. Each row represents the elements of the output for one test case. The shaded squares are assigned to one input list and the blank squares to the other list. Block sizes tested include all powers of two between 1 and n.

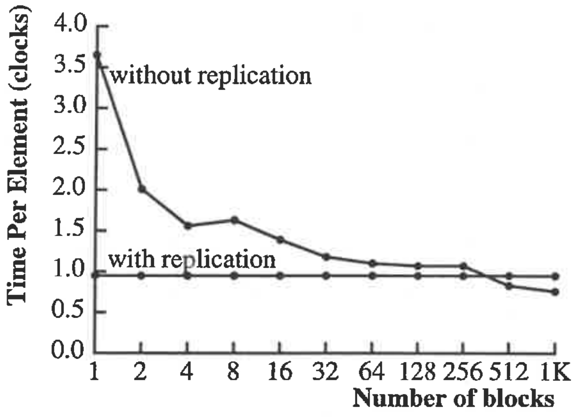
The replication is effective at preventing the worst-case contention, but adds approximately 20 percent to the best-case running time. A more efficient way to implement the replication is to use segmented plus scans to determine where to place copies and segmented copy scans to distribute the data. (These scan operations are described in Section 3.3.3.) In the worst case, p blocks must be replicated. To reduce the replication cost, the amount of replication can be determined using the (d, x)-BSP model, as discussed in Section 3.2.2.

5.4.2 Measuring contention

We designed a test set to capture two classes of inputs for merging algorithms: random test cases and regular cases that can be used for varying the contention. In the random test cases, half of the elements of the destination list C, chosen at random, are assigned to input list A and the remaining elements are assigned to input list B. We generate the regular test cases using block cyclic distributions. Two notable special cases are a block size of n, in which all elements of A are less than all elements of B, and a block size of one in which the elements of C are "shuffled" to A and B. See Figure 5.6. Large block sizes can cause high contention because several virtual processors may point to the same element of an input list during parallel merging. In the worst case (one block of size n, as depicted in Figure 5.5(b)), the expected time per element is $d/2$. If we predict the running time based on the QRQW model (i.e., ignoring network contention and module map contention), increasing the number of blocks is expected to reduce the time linearly until contention is insignificant. As indicated in Figure 5.7(a) the actual performance is slightly worse than the predictions because the injection order is not randomized.



(a)



(b)

Figure 5.7: **(a)** Predicted and measured performance on the CRAY J90 ($p = 4$) for large lists on block-cyclic test inputs. Predictions, shown with dotted curves, do not account for module map contention. The curve labeled *plus* is the sum of the computation term and contention term, whereas the *max* curve is the maximum of these two terms. For b blocks, the predicted memory contention is $n/(2b)$ and the cost per element for contention is $d/(2b)$ ($7.0/b$ on the CRAY J90). **(b)** Effect of replication in merging on 8 processors of the CRAY C90 using block cyclic test inputs. With replication, the time is independent of the number of blocks.

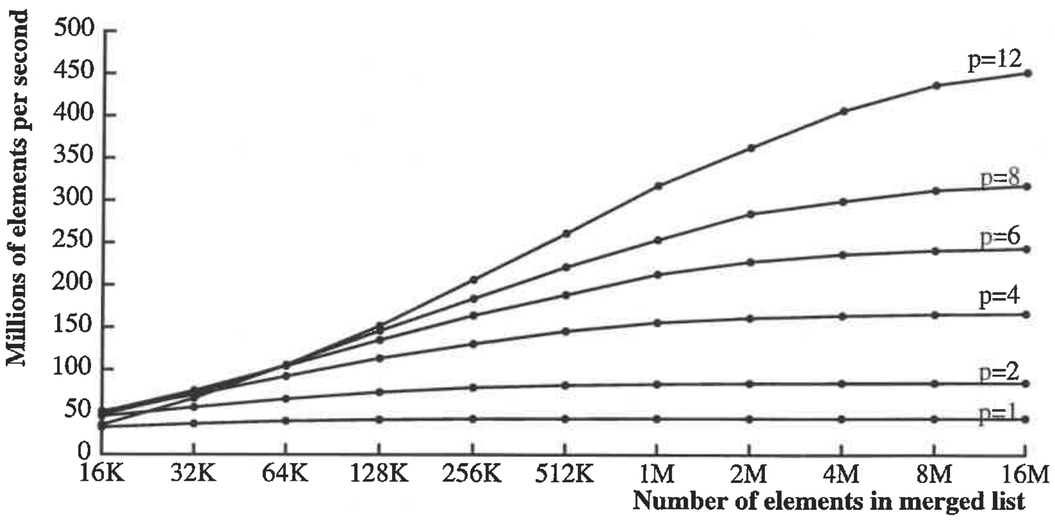


Figure 5.8: Multiprocessor merging performance on the CRAY C90.

5.5 Discussion

Our results can be summarized as follows:

- The running time can be accurately modeled in terms of machine and problem parameters, and the performance model can be used to optimize latency-hiding parameters.

- Contention is not expected to be significant for most input sets, but measurements of the worst-case contention can be predicted fairly accurately, and various replication strategies can be used to avoid slowdowns from contention.

- For large problem sizes, the PRAM merging algorithms are significantly faster than serial merging and variants of bitonic merging. In addition, the PRAM algorithms require less memory. The implementation of bitonic merge requires n words of temporary memory, whereas the PRAM merging algorithms require only $O(vp)$ words of temporary memory. That is, for a fixed set of machine parameters, the amount of temporary memory required is independent of the problem size.

- Implementations of the PRAM algorithms achieve good absolute performance and speedup. Figure 5.8 shows the multiprocessor merging performance on the CRAY C90. For small problem sizes (up to a few thousand elements), there is no benefit from using multiple processors. As the problem size is increased, the speedup improves, and when the problem size reaches millions of elements, the speedup is approximately linear in the number of processors. Using 12 processors, the peak performance on the CRAY C90 is over 450 million elements per second.

Chapter 6

Sparse Matrix Multiplication

This chapter considers the problem of multiplying an arbitrary sparse matrix by a dense vector. More precisely, the problem is to compute $y = Ax$, the product of a matrix $A = (a_{ij}, 0 \leq i < m, 0 \leq j < n)$ and a vector $x = (x_j, 0 \leq j < n)$ yielding the vector $y = (y_i, 0 \leq i < m)$. The number of nonzero elements in the matrix is denoted e, and in the problems of interest, e is only a small fraction of $m \cdot n$. Using a compact representation of the matrix, sparse matrix multiplication requires $O(e + m)$ work.

In addition to having many practical applications, sparse matrix multiplication also serves as a vehicle for studying irregular computation on high-bandwidth multiprocessors. Sparse matrix multiplication can be formulated as a graph algorithm, in which graph edges correspond to matrix elements, graph nodes correspond to vector elements, and the computation consists of weighted neighbor summing. Sparse matrix multiplication is an irregular computation, because access memory access patterns will depend on the sparsity pattern, i.e., the underlying graph structure. For large matrices, there is abundant parallelism inherent in the problem, but the irregularity of the computation, particularly for handling arbitrary sparsity patterns, requires high-bandwidth fine-grained memory access and poses several implementation challenges.

The main focus of this chapter is the implementation of latency-tolerance for *arbitrary* sparse matrices. We introduce a technique for sparse matrix multiplication called SEGMV, which is based on an efficient implementation of *segmented sums*. A segmented sum treats an array as a vector partitioned into contiguous segments, each potentially of a different size, and sums the values within each segment [23]. We compare the performance on the CRAYof the SEGMV algorithm to that of various other algorithms, using a test suite of sparse matrices from the Harwell-Boeing collection and industrial application codes. The performance results show that the implementation of a general-purpose PRAM algorithm is predictable and fast regardless of the underlying structure of the matrix. In addition to measuring times for matrix multiplication on the test matrices, we used SEGMV as the core of the NAS Conjugate Gradient benchmark [15, 16]. On 16 processors of the CRAY C90, an implementation of the benchmark using SEGMV achieves 3.5 Gflops.

The remainder of the chapter is organized as follows. Section 6.1 presents a brief overview of commonly used sparse matrix data structures and algorithms, with an emphasis on work and latency. Section 6.2 quantifies the effect of concurrent reads using the $(d, \mathbf{x})$-BSP and describes several methods of reducing contention using replication, and Section 6.3 describes the SEGMV algorithm, and Section 6.4 compares the performance of SEGMV to other techniques, using a collection of real-life test problems run on a CRAY C90, and Section 6.5 discusses applications and generalizations.

This research was performed jointly with Guy Blelloch and Mike Heroux and was originally published as a technical report [33].

6.1 Background and previous work

In comparing various approaches for implementing sparse matrix-vector multiplication, there are a number of practical issues to consider including the running time for sparse matrix multiplication, the generality of the approach, the preprocessing time for creating the data structures, the convenience of the data structures for performing other sparse matrix computations, the memory required for representing the sparse matrix, and the temporary memory used during the sparse matrix multiplication.

The key focus of most work on sparse matrix multiplication is running time, which for many machines places the focus on reducing the memory bandwidth requirements. For example, previous work on sparse matrix multiplication in the BSP model [21] shows that structure independent computation requires extremely high communication performance and that exploiting the underlying structure of the matrix is of primary importance for achieving efficient implementations.

The generality of an implementation is determined primarily by its ability to hide latency for arbitrary irregular matrices. In the BSP model, hiding the memory latency is conceptually straightforward, since each processor has the addresses of all vector elements that will be accessed in a superstep. However, in addition to hiding memory latency, an efficient implementation must pipeline arithmetic operations in the presence of differing, and potentially very short, row and column lengths. Depending on the type of pipelined-memory machine, it may be important to make use of long vector instructions, to reduce the number of branches executed, or to reduce thread scheduling overhead. Previous techniques for vector computers have optimal latency hiding and work complexity only for restricted classes of matrices. In contrast, SEGMV is insensitive to the matrix structure and therefore hides memory and arithmetic unit latency for matrices that are very irregular.

In the following discussion, we separate previous techniques into specialized techniques (techniques that work best for structured matrices), general techniques (techniques that work for sparse matrices with any structure), and hardware techniques (techniques that have been suggested for direct implementation in hardware).

6.1.1 Specialized techniques

Sparse diagonal systems: Standard discretizations of partial differential equations typically generate matrices that have only a few non-zero diagonals. An appropriate data structure for this type of structured sparse matrix stores only these non-zero diagonals together with offset values to indicate where each diagonal belongs in the matrix (see the DIA data structure in [171, 100]).

Feature extraction: Another approach to optimizing sparse matrix operations is to decompose the matrix into additive submatrices and store each submatrix in a separate data structure. For example, the Feature Extraction Based Algorithm (FEBA) presented by Agarwal et al. [3] recursively extracts structural features from a given sparse matrix structure and uses the additive property of matrix multiplication to compute the matrix-vector product as a sequence of operations. The FEBA scheme first attempts to extract nearly dense blocks. From the remainder of entries, it then attempts to extract nearly dense diagonals and so on. This is a powerful technique, especially on machines where the performance of dense computations is substantially better than that of general sparse computations. However, one is required to have an extra copy of the original matrix with the potential for a non-trivial amount of fill-in. In addition, pre-processing time can be significant.

6.1.2 General techniques

General-purpose sparse matrix data structures for are appropriate in cases where the matrix has no known regular sparsity pattern or where a general-purpose data structure is needed in order to handle a variety of sparse matrix patterns. We review only a few of the many general-purpose sparse data structures [3, 71, 74, 76, 100, 119, 142, 153, 157, 158, 171, 177].

Compressed sparse row: One of the most commonly used data structures is the compressed sparse row (CSR) format. The CSR format stores the entries of the matrix row-by-row in a scalar-valued array VAL. A corresponding integer array INDX holds the column index of each entry, and another integer array PNTR points to the first entry of each row in VAL and INDX. For example, for a sparse matrix

$$A = \begin{pmatrix} 11 & 0 & 13 & 14 & 0 & 0 \\ 0 & 0 & 23 & 24 & 0 & 0 \\ 31 & 32 & 33 & 34 & 0 & 36 \\ 0 & 42 & 0 & 44 & 0 & 0 \\ 51 & 52 & 0 & 0 & 55 & 0 \\ 61 & 62 & 0 & 64 & 65 & 66 \end{pmatrix},$$

the CSR representation of A consists of the following three arrays:

$$\begin{array}{lll} \text{VAL} & = (& 11\ 13\ 14\ 23\ 24\ 31\ 32\ 33\ 34\ 36\ 42\ 44\ 51\ 52\ 55\ 61\ 62\ 64\ 65\ 66), \\ \text{INDX} & = (& 1\ 3\ 4\ 3\ 4\ 1\ 2\ 3\ 4\ 6\ 2\ 4\ 1\ 2\ 5\ 1\ 2\ 4\ 5\ 6), \\ \text{PNTR} & = (& 1\ 4\ 6\ 11\ 13\ 16\ 21\ ). \end{array}$$

Using this CSR format, the sparse matrix-vector product operation, $y = Ax$, can be written as a loop over the rows of A in which each iteration computes an inner product of row i with the vector x:

```
DO I = 1,M
   Y(I) = 0.0
   DO K = PNTR(I), PNTR(I+1)-1
      Y(I) = Y(I) + VAL(K)*X(INDX(K))
   END DO
END DO
```

Note that the trip counts on the inner loop are determined by the number of entries in each row, which is typically small and does not grow with the problem size. Thus, the performance of this kernel on vector machines is usually poor, due to limited latency-hiding on the inner loop.

The CSR format and its Compressed Sparse Column (CSC) analogue are the most common general-purpose data structures. They are flexible and easy to use for many matrix operations. However, for most matrix operations, their performance on vector multiprocessors is suboptimal because of limited tolerance to memory latency. To achieve higher performance, loop interchange can be used in a variety of forms to operate on several rows at once. Because each row can be a different length, special data structures are often used to make the computation more regular. Two examples are the Ellpack/Itpack data structure and the Jagged Diagonal data structure.

Ellpack/Itpack: The Ellpack/Itpack (ELL) storage format [120] forces all rows to have the same length as the longest row by padding with zeros. Since the number of rows is typically much larger than the average row length, the sparse matrix-vector multiplication can be efficiently vectorized by expressing it as a series of operations on the columns of the modified matrix. The ELL format is efficient for matrices with a maximum row size close to the average row size, but the effective computational rate can be arbitrarily bad for matrices with general sparsity patterns because of wasted computation. In addition, memory space is wasted by storing extra zeros, and, in some applications, by having to maintain an unmodified version of the matrix.

Jagged diagonal: The Jagged Diagonal (JAD) format [7] is also designed to exploit vectorizable column operations, but without performing unnecessary computation. In the JAD format, rows are permuted into order of decreasing length. The leftmost elements of each row (with their column indices) are stored as a dense vector, followed by the elements second from the left in each row stored as a (possibly shorter) dense vector, and so on. Sparse matrix-vector multiplication proceeds by operating on each column in turn, decreasing the vector length as the length of the current column decreases. Sparse matrix-vector multiplication using the JAD format performs well on vector multiprocessors for most matrices. However, performance can be suboptimal in some cases, for example, when there are only a few long rows. The main disadvantage of the JAD representation is that constructing the data structure requires permuting the rows of the matrix. Thus, it is difficult to construct and difficult to modify, and in addition, makes expressing a standard forward/back solve or other operations difficult.

6.1.3 Hardware techniques

Taylor et al. [193] proposed a modification to the compressed sparse column (CSC) data structure that stores the non-zero elements of each column with an additional zero value placed between each set of column values. With this data structure, sparse matrix-vector multiplication can be expressed as a single loop over all non-zero elements, rather than a nested loop over columns and rows. Each iteration of the long vector loop tests the current matrix value for equality with zero and either continues accumulating or updates the current column index. The authors note that the conditional statement "cannot be executed efficiently on conventional vector machines because the vector length is not given explicitly" and propose special-purpose zero-detection hardware to overcome this problem.

6.2 Contention

6.2.1 Measuring and modeling contention

We constructed an experiment to measure the effect of contention when multiplying a sparse matrix by a dense vector using SEGMV. For the purposes of analyzing contention, the most important characteristic of the implementation is that elements from the input vector are gathered based on the column indices of the nonzero matrix elements. Elements of the input vector are typically read multiple times. Thus, our formulation of sparse matrix multiplication can be viewed as a CREW algorithm or viewed as a QREW algorithm in which the maximum contention is equal to the number of elements in the densest column. A dense column can arise in practice from having a global constraint or bias, such as a circuit with many connections to ground.

In the experiment, we constructed a set of test matrices with one dense column and an average row length of seven. The number of rows and the total amount of work are held constant, while the number of elements in the dense column is varied. Except for elements in the dense column, column indices are selected at random.

The predicted running time for sparse matrix multiplication on the (d, x)-BSP is:

$$\max\left(\frac{w}{p}, d \cdot R\right) ,$$

where w is a constant that accounts for the combined effect of the work and the gap, and the total contention R is equal to the maximum column length c_{max}. Figure 6.1 shows measured and predicted times as a function of the length of the dense column, using the measured value $w = 4.8$ for the CRAY J90. The figure demonstrates the importance of accounting for the memory bank delay and clearly shows that worst-case contention is significant. Most matrices used in practice will not cause significant contention. However, as described later in Section 6.4.4, we did observe significant degradation from contention on one matrix in our test suite (FIFE1Q).

6.2.2 Reducing contention

The cost of memory contention can be reduced using a variety of replication strategies:

input vector replication: The entire input vector can be replicated a fixed number of times, as described in Section 3.2.2. This strategy is simple to implement and avoids worst-case contention, but requires additional work for each sparse matrix multiplication, requires additional memory, and does not eliminate contention costs.

segmented scan replication: Concurrent reads can be eliminated entirely by using segmented scans [36]. Each vector element, x_i, is replicated c_i times, where c_i is the number

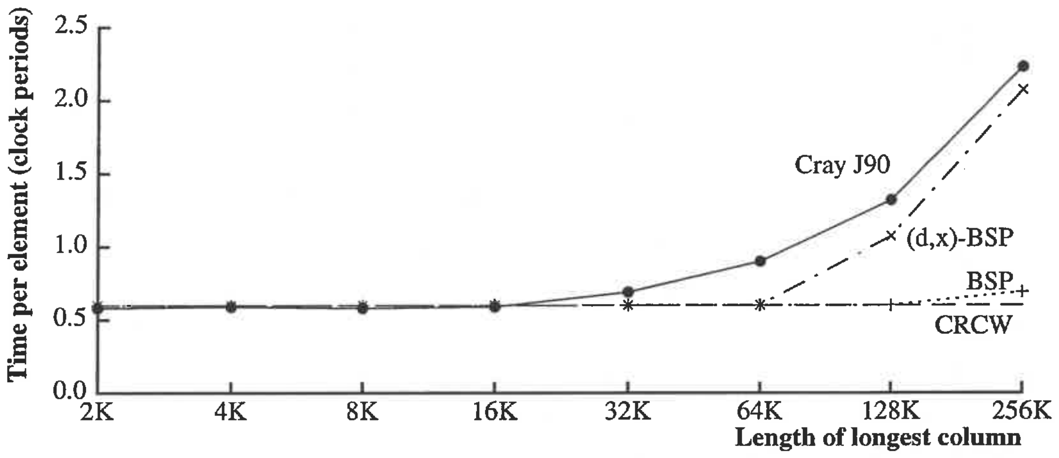


Figure 6.1: Time per nonzero element for sparse matrix vector multiplication, on matrices with one dense column and an average row size of seven. All matrices have 256K rows. Measured times are given for an 8-processor CRAY J90, and predicted times are given for the (d, x)-BSP, BSP, and CRCW.

of nonzero elements in the ith column. This approach eliminates contention and is asymptotically optimal, but is approximately a factor of two slower than the CREW algorithm when the contention is insignificant.

(d, x)-BSP replication: In the (d, x)-BSP model, when the contention k for e memory requests satisfies the condition $e \leq \frac{sg}{pd}$, the contention term will not affect the running time. For example, in Figure 6.1, the contention does not contribute to the running time until the maximum contention reaches tens of thousands, at the "knee" of the curve. Using the (d, x)-BSP, replication parameters can be determined such that there is just enough replication for each element to avoid any penalties from memory contention. In applications of sparse matrix multiplication in linear system solvers, the elements of the input vector change each iteration, but the same matrix is used multiple times. Thus, determining the optimal replication parameters is essentially an "off-line" computation, and the cost can be amortized over many sparse matrix multiplications.

6.3 Cray implementation

Our approach to sparse matrix multiplication combines several of the advantages of the previous approaches discussed in Section 6.1. The data structure we use is an augmented form of the CSR format. The computational structure of our algorithm is similar to ELL and JAD in that we get parallelism from operating on many rows at once. But rather than using a heuristic approach to making the computation more regular, we reformulate sparse matrix-vector multiplication in terms of segmented vector operations. This approach is similar to the technique proposed by Taylor et al., since their data structure is one method for representing segmented vectors, and their algorithm for sparse matrix-vector multiplication using a single loop is essentially a segmented vector operation. However, our approach does not require special-purpose hardware because we are able to express the conditional computation in terms of standard vector merge instructions. By combining features of previous approaches in this way, we obtain an algorithm for vector multiprocessors that is insensitive to matrix structure, uses a convenient format, requires minimal preprocessing, exposes a large degree of parallelism (without performing unnecessary computation) and does not require special-purpose hardware. Section 6.3.1 explains how segmented scans can be used for sparse matrix multiplication, and our algorithm is described in detail beginning in Section 6.3.2.

6.3.1 Using segmented scans for sparse matrix multiplication

Chapter 3 provides background on segmented scans. Here we consider how to use segmented scans for sparse matrix multiplication. A sparse matrix can be represented as a segmented vector by treating each row of the matrix as a segment. Since the CSR (compressed sparse row) format is already organized in such segments, the only additional structure required an array of flags, which can be generated from pointers to the start of each row (PNTR). For example, the matrix in Section 6.1.2 the flags would be organized as:

```
VAL  = ( 11 13 14 23 24 31 32 33 34 36 42 44 51 52 55 61 62 64 65 66 ),
FLAG = (  T  F  F  T  F  T  F  F  F  T  F  T  F  F  T  F  F  F  F ).
```

Using these flags, sparse matrix multiplication can be expressed in HPF as follows:

```
PRODUCTS = VAL * X(INDX)
SUMS = SUM_SUFFIX(PRODUCTS,SEGMENT=PARITY_PREFIX(FLAG))
Y = SUMS(PNTR)
```

This code first calculates all the products by indirectly accessing the values of X using INDX and multiplying them by VAL. The second line uses the SUM_SUFFIX function.[1] The SUFFIX functions in HPF execute the scan backwards, such that each element receives the sum of values that follow it, rather than precede it. The result of the SUM_SUFFIX on the PRODUCTS array leaves the sum of each segment in the first element of the segment.

For example:

[1] HPF defines segments using an "edge-triggered" representation that marks segment boundaries by transitions from TRUE to FALSE or from FALSE to TRUE in the FLAG array. The conversion to HPF segments is performed with the PARITY_PREFIX intrinsic, which executes a scan using exclusive-or as the binary operator.

$$\begin{array}{lcl}
\texttt{PRODUCTS} & = & (\; 5 \quad 1 \quad 3 \quad 4 \quad 3 \quad 9 \quad 2 \quad 6 \;) \\
\texttt{FLAG} & = & (\; \texttt{T} \quad \texttt{F} \quad \texttt{T} \quad \texttt{F} \quad \texttt{F} \quad \texttt{F} \quad \texttt{T} \quad \texttt{F} \;) \\
\texttt{SUM\_SUFFIX(PRODUCTS, SEGMENT)} & = & (\; \overline{\boxed{6} \quad 1} \quad \overline{\boxed{19} \quad 16 \quad 12 \quad 9} \quad \overline{\boxed{8} \quad 6} \;)
\end{array}$$

Finally, the sums are gathered from the first element of each segments using the PNTR array.

Using an implementation based on segmented scans gives reasonable running times on vector multiprocessors and massively parallel machines [195]. Our experiments show that on the CRAY C90, an implementation of this algorithm based on optimized segmented scans [48] is between 1.2 and 2.0 times slower than the Jagged Diagonal method (JAD). Because of the simplicity of the code, the use of CSR format rather than a permuted data structure, and the much lower cost of setup, this might warrant using the scan-based method in many cases. However, as described in Section 6.3.2, we can modify this algorithm such that it is almost always as fast or faster than JAD but still maintains these advantages.

The scan-based algorithm is slower than JAD mainly because it performs unnecessary work; the scan generates all the partial sums within each segment, but only the sums at the beginning of each segment are used. Both the parallel and vector scan algorithms must pass over the data twice and execute a total of $2(e-1)$ additions instead of the $e-1$ required by a serial algorithm. However, if only the sum of each segment is needed, only one pass over the data is required, executing a total of only e additions. To enable this optimization, we need to generate some additional information in a setup phase and pass this extra data to our segmented summing operation. We generate a *segment descriptor* from the PNTR array of length `M` and the total number of non-zero elements `N`. It returns an array of integers; the format of this array will be described in the next section.

6.3.2 Segmented sum algorithm

The intuition behind the segmented summing algorithm is that segmented vectors can be broken into equally sized blocks, independent of where segment boundaries are located. Then, working on each block simultaneously, the algorithm find the sum of each segment that is completely contained within a block, or the partial sums of each segment that spans multiple blocks. Finally, in a post-processing step, partial sums are added for each piece of the segment that spans multiple blocks.

To parallelize this algorithm, we treat one-dimensional arrays as two-dimensional arrays, where the notion of "block" corresponds to a column in the two-dimensional representation. Specifically, we treat the VAL and FLAG arrays of size e as two-dimensional arrays (in column-major order) with v columns and $s \approx e/v$ rows. (This conceptual layout is used to simplify the presentation and does not require any data motion.) The parameter v, the number of virtual processors, is chosen to be approximately equal to the product of the number of processors and

```
VAL =  .5 .2 .3 .1 .4 .2 .1 .2 .1 .5 .1 .3 .2 .2 .4 .2 .4 .5 .3 .4 .4 .1 .6 .2 .5 .4 .3 .6 .3 .5
FLAG = T  F  T  F  F  F  T  F  F  F  T  F  F  F  T  F  F  T  F  F  F  F  F  F  F  T  F  T  T  F
```

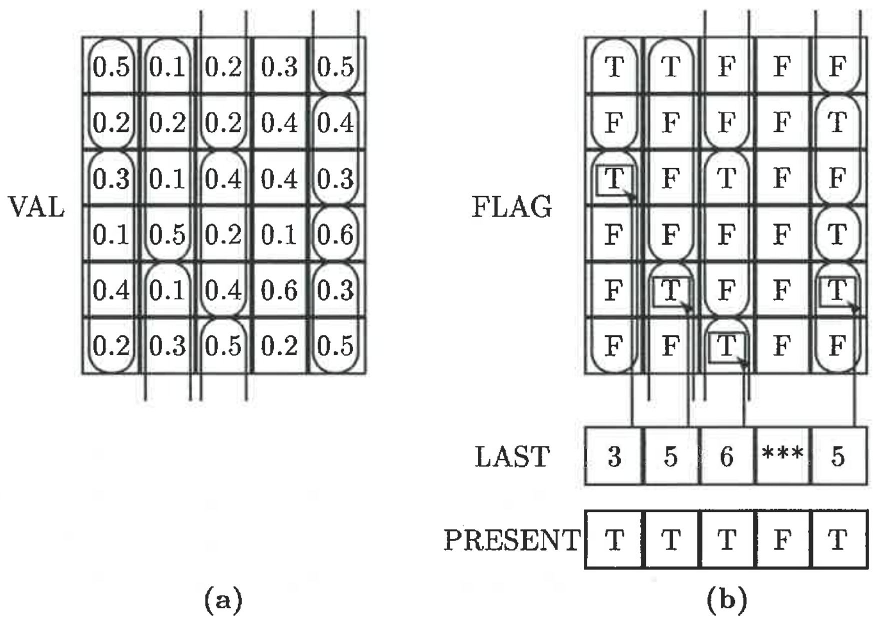


Figure 6.2: (a) The values in the segmented vector VAL in two-dimensional form with $v = 5$ columns and $s = 6$ rows. (b) The elements of the FLAG array are TRUE at the beginning of a segment and FALSE elsewhere. Each column has a logical variable PRESENT that indicates whether any segments start within the column and an index LAST pointing to the last TRUE flag in the column.

the hardware vector length so that we can express the algorithm in terms of vectorizable row operations. Figure 6.2 shows an example of a segmented vector in this form. Note that some segments, which we call *spanning segments*, span multiple columns.

To implement SETUP_SEG we need to pre-compute some information about the structure of the segments. The preprocessing consists of the following steps (depicted in Figure 6.2(**b**)):

1. As with the setup for segmented scans, set the FLAG elements to TRUE at the beginning of each segment, and FALSE elsewhere.

2. Determine for each column whether any segments start in that column and place the result in the logical array PRESENT(1:v). Place the row index of the last segment beginning in each column (if any) in LAST(1:v).

The result of SEG_SUM is a segment descriptor containing the LAST, PRESENT, and FLAG arrays. (To save memory, the flags can be represented using packed bit fields stored in an integer array.) Once we have this information, the SEG_SUM algorithm works in three stages:

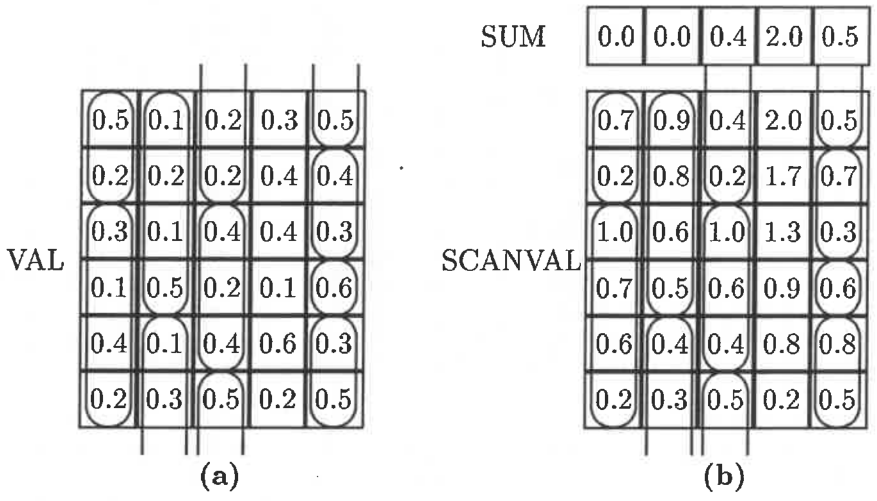


Figure 6.3: **(a)** The representation of the sparse matrix VAL with $v = 5$ columns and $s = 6$ rows. **(b)** The result of performing a backwards segmented scan in each column. For segments contained entirely within a column, the value at the beginning of each segment holds the sum of the segment.

1. Perform a backwards segmented scan within each column. Place the sum of the first partial segment in each column into SUM. (See Figure 6.3.) For all segments that are contained within a single column, the sum of the segment will be in the first element of the segment.

2. Find the sum of spanning segments by adding the sum of each piece (See Figure 6.4):

   (a) Perform a backwards segmented scan on SUM into SCANSUM in order to compute the adjustment that must be made to each spanning segment.

   (b) For each spanning segment, increment the first element in the segment by the value of SCANSUM in its column.

3. Gather the sum of each segment from its first element.

6.3.3 Sparse matrix-vector multiplication

As discussed in Section 6.3.1, sparse matrix multiplication can be expressed in terms of a segmented sum primitive. However, we can build a direct implementation of sparse matrix multiplication by replacing all uses of the source array (VAL) in the columnwise scan with `VAL * X(INDX)`, which first multiplies the matrix element by the corresponding vector element.

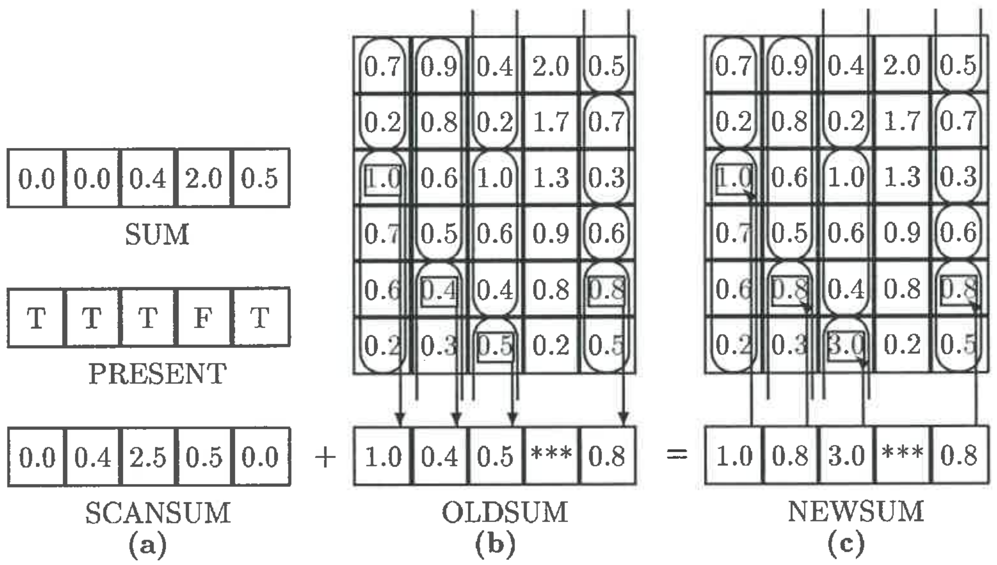


Figure 6.4: In Figure (a) the SCANSUM array is computed by performing a backwards segmented scan on SUM using the PRESENT flags as segment boundaries. Figures (b) and (c) show how in each column, the first value in the last segment is incremented by SCANSUM to produce the sum of the segment.

On many machines, a direct implementation of sparse matrix multiplication is more efficient because of reduced memory traffic and greater opportunities for chaining memory and arithmetic operations.

6.3.4 Implementation details

The structure of the SEGMV column scan is similar to the Ellpack/Itpack and Jagged Diagonal kernels, except that the accumulated sums are reset to zero when a segmented boundary is encountered. This conditional operation can be vectorized using a vector merge instruction. To reduce the memory-bandwidth and storage requirements, the flags are represented using packed bit-masks (up to 64 per word) stored in the form used by the vector mask register. By using packed flags we are able to implement the conditional instruction in the columnwise scan at almost no additional cost, since loading the flags is virtually free, and the merge instruction chains with the multiplication and addition in the hardware pipeline.

We implemented SEGMV for the CRAY C90 in Cray Assembly Language [58] (CAL) using microtasking [59] from C to take advantage of multiple processors. We coded in assembly language to get maximal chaining and reuse of operands. However, to enhance portability and to reduce the time for program development, we used a macro-assembler (written in LISP) with support for register naming, loop unrolling, and handling packed masks. We also implemented a version of SEGMV in Fortran (using packed flags) that performs approximately a factor of two slower than the CAL version for large matrices. The Fortran version was used for experimenting with the extensions to the algorithm mentioned in Section 6.5.1. Further implementation details are described in a technical report [33].

6.4 Performance analysis

In this section we present performance results for sparse matrix multiplication using a 16-processor 1024-bank CRAY C90 at Cray Research Inc. in dedicated mode. The test suite (provided by Mike Heroux at Cray Research), contains sample problems from the Harwell-Boeing test set [72] and industrial application codes. We also present a performance model for the SEGMV implementation that accurately predicts its running time.

6.4.1 Test suite

Table 6.1 presents statistics and descriptions of our test problems, and Figure 6.5 shows the distribution of problem sizes and row sizes in the test suite. We chose a representative collection of matrices from several important application areas. Most of them are from the standard Harwell-Boeing collection [72] and will facilitate comparison with other results. We also included several large nonsymmetric matrices, because the Harwell-Boeing collection does not contain a large set of nonsymmetric matrices.

6.4.2 Results for matrix multiplication

Table 6.2 and Figure 6.6 report timing information comparing our implementation to several other implementations on the set of test matrices. The JAD implementation is a highly-optimized Cray Research library routine implemented in Cray Assembly Language (CAL). The ELL, CSR, and CSC implementations are based on optimized Fortran kernels. Our SEGMV routine is coded in CAL. In all cases, even if the matrix is symmetric, all non-zero elements are stored. (Symmetric storage is not used in this study because efficient methods have not been developed for processing symmetric matrices with the Ellpack/Itpack, Jagged Diagonal, or SEGMV data structures.) The Flop rate is computed based on the running time t for one sparse matrix-vector multiplication (not including the setup computation), the number of equations m, and the total number of non-zero elements e. Matrix-vector multiplication requires e multiplications and $e - m$ additions, yielding 64-bit Mflop rate of $(2e - m)/t$.

On one processor, SEGMV outperforms all other implementations on large, irregular test matrices, as shown in Figure 6.6. On multiple processors, SEGMV outperforms the other im-

Problem	Entries	Dim	Ave	Max	Sym	Description
Structures Problems (Finite Element)						
BCSSTK15	117816	3948	29.8	44	$\surd$	Stiffness matrix for module of an Off-shore Platform
BCSSTK30	2043492	28924	70.7	219	$\surd$	Stiffness matrix for Off-shore Generator Platform (MSC NASTRAN)
NASASRB	2677324	54870	48.8	276	$\surd$	Structure from NASA Langley, Shuttle Rocket Booster
Standard Finite Difference Operators						
GR-30X30	7744	900	8.6	9	$\surd$	Symmetric matrix from nine point start on a 30 by 30 grid
CUBE12	982600	8640	113.7	135	$\surd$	Symmetric pattern, 27 point operator on 12 by 12 by 12 grid
Oil Reservoir Problem						
SHERMAN2	23094	1080	21.4	34		Thermal Simulation, Steam injection, 6 by 6 by 5 grid, five unknowns
Computational Fluid Dynamics Problems						
BBMAT	1771722	38744	45.7	132		Beam + Bailey 2D Airfoil Exact Jacobian: mach = 0.08, angle = 2.0
3DEXT1	9045076	118564	76.3	108		3D Fully-coupled Incompressible NS Polymer Extrusion Problem
Chemical Process Simulation Problem						
FIFE1Q	392188	47640	8.2	44085		Chemical Process Simulation Test Matrix (Permuted)

Table 6.1: Statistics for the test matrices: number of nonzero entries (e), the dimension (m), average row size, maximum row size, and a brief description of each problem. For those matrices that are symmetric (indicated by the *Sym* column), the matrix statistics describe the full nonsymmetric representation.

plementations by a greater margin, with an absolute performance on 16 processors of over 3.7 Gflops.

6.4.3 Performance model

One advantage of the SEGMV algorithm is that its running time algorithm is predictable. This section presents a performance model for the SEGMV algorithm with equations broken down into components for columnwise loops over the e matrix values, loops over each of the m segments, loops over each of the p processors, and contention costs. All constants were measured on the CRAY C90 and are expressed in 4.167 nsec clock periods:

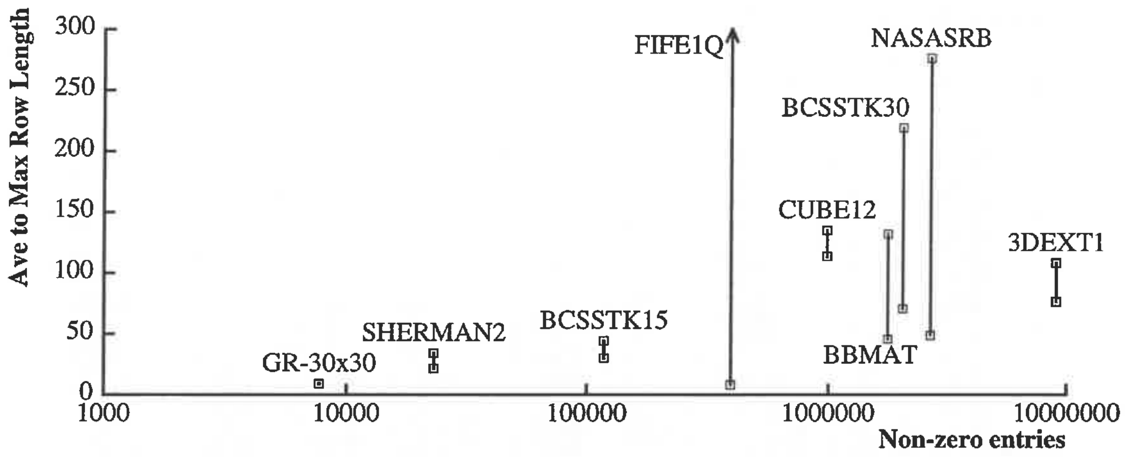


Figure 6.5: Statistics for the test matrices. Each test matrix is represented by a vertical line. The horizontal position of the line indicates the number of non-zero entries in the matrix. The bottom point indicates the average row length and the top point indicates the maximum row length. (The FIFE1Q matrix has a maximum row length of 44085, which is represented as an arrow.) The test suite includes a wide variety of row sizes and matrix sizes (ranging from under 10^4 nonzeros to nearly 10^7 nonzeros).

	1 CPU					4 CPU					16 CPU				
Matrix	CSR	CSC	ELL	JAD	SEG	CSR	CSC	ELL	JAD	SEG	CSR	CSC	ELL	JAD	SEG
GR-30X30	15	26	333	336	278	48	86	784	784	471	163	141	927	900	205
SHERMAN2	34	65	117	248	288	121	216	386	501	795	419	364	1017	1314	488
BCSSTK15	45	84	205	312	304	164	293	746	893	1056	617	551	2303	2792	2479
FIFE1Q	14	25		165	249	50	88		284	577	199	164		436	611
CUBE12	126	182	153	163	322	476	645	579	540	1166	1816	1715	2042	1954	3758
BBMAT	63	118	103	255	311	235	422	396	389	1115	932	910	1270	1191	3639
BCSSTK30	86	157	90	168	312	323	561	335	446	1162	1246	1260	1157	1411	3749
NASASRB	67	123	65	224	311	249	444	241	739	1192	986	907	789	2117	3740
3DEXT1	92	165	187	244	314	345	590	709	616	1165	1357	1297	2433	2260	3797

Table 6.2: Performance results (Mflops) on the CRAY C90 comparing five implementations of sparse matrix multiplication: Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Ellpack/Itpack (ELL), Jagged Diagonal (JAD), and SEGMV (SEG). The test matrices are listed in order of increasing number of non-zero entries. Figure 6.6 shows this data using bar charts. (On the FIFE1Q matrix, no results are given for ELL because the data structure would require over 2 Gwords.)

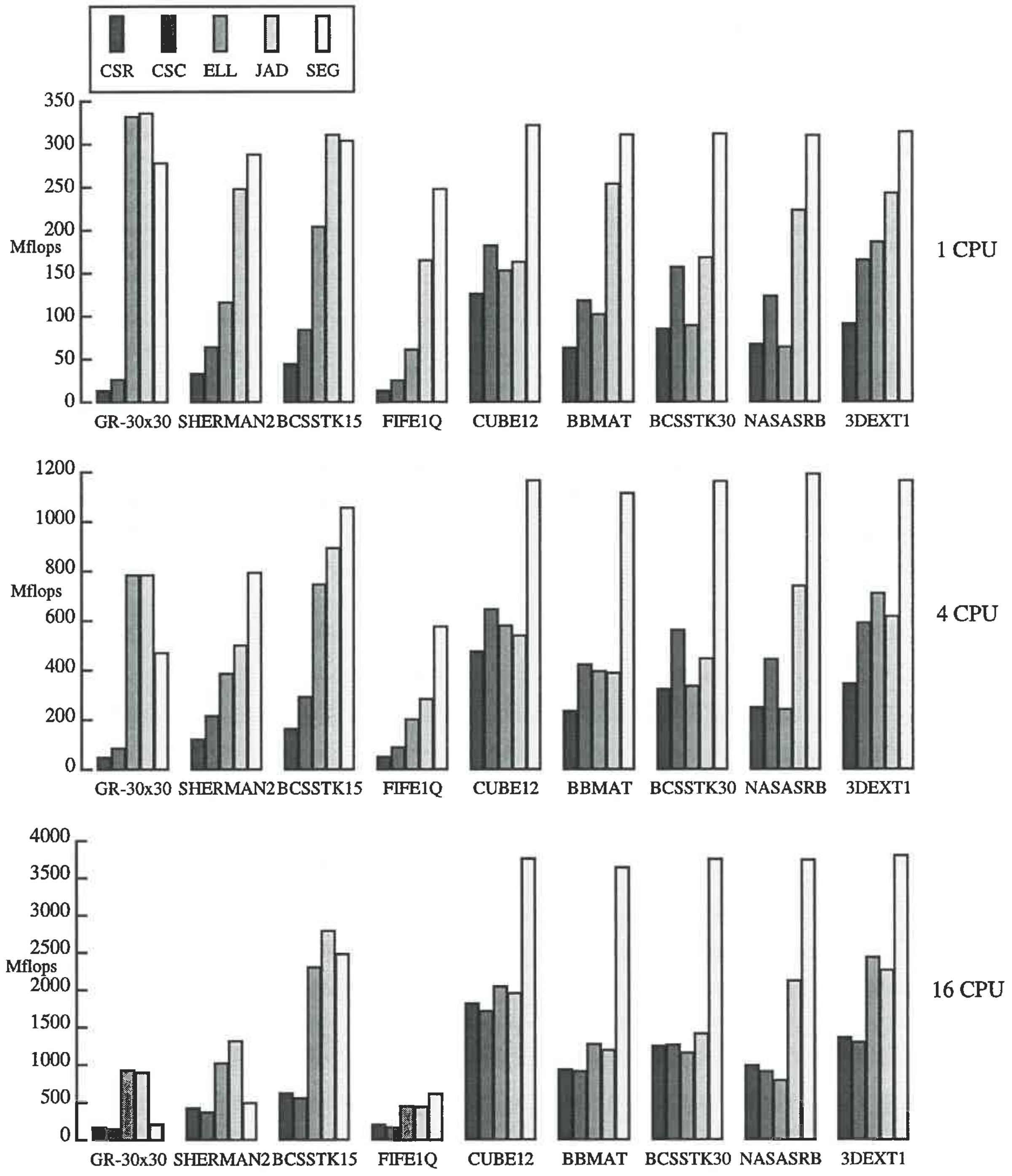


Figure 6.6: Performance results on a CRAY C90 comparing implementations of sparse matrix multiplication. The test matrices are shown in order of increasing number of non-zero entries. The SEGMV implementation outperforms all other implementations on the large irregular test matrices.

The setup time is given by the following equation:

$$t_{\text{setup}} = \frac{t_f \cdot e + t_s \cdot m}{p} + t_o$$

where the constants t_f, t_s, and t_o are defined as follows:

$$\begin{aligned} t_f &= \text{time per element for columnwise loops} \approx 1.7 \\ t_s &= \text{time per element to mark segment starts} \approx 1.3 \\ t_o &= \text{constant overhead} \approx 1000 \end{aligned}$$

The time to perform one matrix-vector multiplication is given by the following equation:

$$t_{\text{mvmult}} = \max\left(\frac{t_c \cdot e + t_g \cdot m}{p} + t_p \cdot p + t_o, d \cdot c_{\text{max}}\right)$$

where the constants t_c, t_g, and t_p are defined as follows:

$$\begin{aligned} t_c &= \text{time per element for columnwise segmented scan loop} \approx 1.5 \\ t_g &= \text{time per element to gather result from SCANVAL} \approx 1.5 \\ t_p &= \text{time per element for across-processor serial scan loop} \approx 110 \end{aligned}$$

and where $d \cdot c_{\text{max}}$ is a contention term computed from the bank delay and the length of the longest column. These equations accurately model the running time from the problem size (e, m, and c_{max}) without additional information about the matrix structure. Predictions for a single CRAY C90 processor are within 15 percent of the measured running times for each matrix in the test suite.

6.4.4 Comparison with other vector implementations

This section compares the running time, setup time, and memory usage of SEG and other algorithms for vectorized sparse matrix multiplication.

Latency tolerance: The performance of previous methods for sparse matrix multiplication can vary greatly with the structure of the matrix, as shown by the results on the test suite. In order to explain some of the underlying reasons for this variation, we generated two sets of test matrices: one set for measuring the dependence on average row size and one set for measuring the dependence on maximum row size. Figure 6.7(a) measures the performance on matrices that contain the same number of elements in each row. As expected, CSR is very sensitive to row size because the row size determines the amount of latency tolerance (i.e., the vector register length). On the other hand, ELL, JAD, and SEG all perform equally well independent of the row size, because their computational structure is nearly identical for matrices with a fixed row

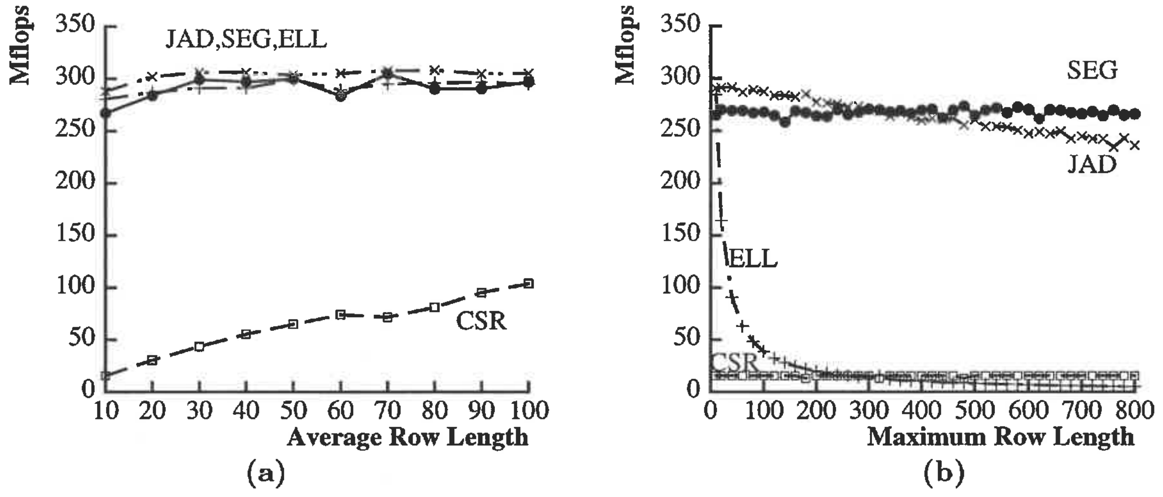


Figure 6.7: **(a)** Performance on one processor of the CRAY C90 as the average row size is varied. Each row contains the same number of non-zero entries, and the total number of non-zero entries is fixed at 10^5. Column indices are generated randomly. The CSR performance varies with the average row length, whereas the other formats are independent of the average row length when all rows are close to the same length. **(b)** Performance as the maximum row size is varied. The number of non-zero entries is fixed at 10^5 and the average row size is fixed at 10. Column indices are generated randomly. As the maximum row size is increased, the ELL performance degrades quickly and the JAD performance degrades slightly.

size. Figure 6.7(b) measures the performance on matrices where all rows are the same length except for one long row. The performance of CSR is insensitive to the maximum row length. The performance with the ELL data structure degrades severely as the maximum row size is increased, because all rows must be padded with zeros to the maximum row size. The JAD data structure exhibits a slight dependence on the maximum row size. Note that the SEGMV performance is independent of the maximum row size and nearly independent of the average row size.

The effects of matrix structure can be seen from the performance results on the test suite. The CSR and CSC implementations perform fairly well on matrices with long rows and columns (such as CUBE12) and poorly on matrices with short rows and columns (such as GR-30X30). The ELL and JAD implementations perform quite well on matrices that have a maximum row length near the average row length (such as GR-30X30 and BCSSTK15).

Contention: One of the most interesting performance results is that ELL, JAD, and SEG all perform poorly on the FIFE1Q matrix. This matrix contains one very dense row and column, which occurs in some application areas, such as chemical process simulation. Because the dense column accounts for over 10 percent of the non-zero matrix elements, a substantial fraction

Matrix	JAD	SEG	ratio
GR-30X30	3	0.1	23
SHERMAN2	4	0.2	16
BCSSTK15	14	0.9	15
FIFE1Q	124	3.1	41
CUBE12	50	7.0	7
BBMAT	161	13.0	12
BCSSTK30	141	14.5	10
NASASRB	219	19.1	11
3DEXT1	606	65.0	9

Table 6.3: Setup times (in msec) for Jagged Diagonal and SEGMV on one processor.

of all memory fetches are made to the same element of the input vector. Without accounting for contention, the expected performance on 16 processors is approximately 3 Gflops. A straightforward calculation using the contention term of the performance model predicts that the performance cannot exceed 670 Mflops, which is consistent with the actual performance of 577 Mflops on 4 processors and 611 Mflops on 16 processors.

Setup time: Execution time for matrix-vector multiplication is not the only important consideration. Large setup times can be significant for certain applications, such as adaptive mesh algorithms. Table 6.3 lists setup times on one processor for SEGMV and Jagged Diagonal. The setup time for SEGMV is approximately the same as the time for a single matrix-vector multiplication. The setup time for Jagged Diagonal is an order of magnitude higher, because generating the data structures requires integer sorting.

Memory usage: The CSR and CSC implementations do not require scratch space. In practice, the JAD and ELL data structures consume additional memory when the user must maintain the original matrix in addition to the permuted or padded version. The SEGMV data structure requires only approximately $e/64$ words for the data structure (in addition to the space for the CSR data structure). The SEGMV algorithm requires e words of scratch space for the SCANVAL temporary array, but we can greatly reduce this requirement by breaking the matrix into a number of strips and processing one strip at a time. This stripmining, which can be hidden from the user, reduces the scratch space required from the size of the matrix to the size of a single strip. The strip size should be chosen to balance the memory consumption with the startup time incurred for processing each strip. For the large matrices in which scratch space is an issue, the stripmining would not significantly affect performance.

Machine	Procs	CG (C90 Equiv.)	Gflops
Cray C90	1	1.0	0.45
Cray C90	16	11.6	5.17
Cray T90	1	1.7	0.74
Cray T90	16	10.8	4.80
Cray J90	1	0.2	0.10
Cray J90	16	2.9	1.28
Fujitsu VPP700	30	9.1	4.08
NEC SX-4	1	1.5	0.67
NEC SX-4	4	4.7	2.09
Cray T3E	1	0.2	0.07
Cray T3E	256	5.5	2.47
IBM RS/6000 SP (P2SC)	8	1.1	0.47
IBM RS/6000 SP (P2SC)	64	5.4	2.42
SGI Origin2000	1	0.1	0.04
SGI Origin2000	31	1.5	0.69

Table 6.4: Performance on the NAS Conjugate Gradient (CG) benchmark (Class B, version 1.0) [173]. The performance is expressed as a throughput measure normalized to the performance of a single-processor CRAY C90 and also converted to Gflops. The CRAY C90, CRAY T90, and CRAY J90 results use our Fortran implementation.

6.4.5 NAS benchmark results

An implementation of the NAS Conjugate Gradient benchmark using our CAL implementation of SEGMV runs at a rate of 325 Mflops on a single CRAY C90 processor, and at 3.5 Gflops on 16 processors. (However, our implementation uses assembly language, which is not permitted for official NAS results.)

We also implemented a Fortran version of the NAS CG benchmark in order to experiment with different aspects of the memory system behavior. We implemented a variant of the Jagged Diagonal method, because it is easier to code in Fortran than SEGMV and is quite effective for the matrix used in the CG benchmark. The matrix is generated with a random structure, has an average row length of well over one hundred, and has a smooth distribution of row lengths. Our implementation makes two optimizations to improve memory system behavior. First, column indices are packed two to a word to reduce the bandwidth demands. Second, the elements within each row are reordered to reduce network and bank contention. Recall from our experiments with the histogram operation used in radix sort (Figure 4.1) that the CRAY C90 is sensitive to the injection order; for the histogram, the CRAY C90 performs approximately twice as fast with a perfect injection order compared to a random order. In the CG benchmark, before beginning the conjugate gradient iterations, the elements in each row of the matrix are reordered using a greedy algorithm, such that the indirect references in each vector gather instruction tend to refer sequentially to different sections in the interleaved memory system. This reordering procedure improves performance by approximately 20 percent. The Fortran CG implementation is faster and more scalable than the previous CRAY C90 implementation [16], with single processor performance 1.3 times better and 16-processor performance 1.7 times better than the previous implementation. Our 16-processor CRAY C90 implementation is the fastest ever reported on the NAS CG benchmark. Results for a variety of vector and microprocessor-base multiprocessors are presented in Table 6.4.

6.5 Applications

6.5.1 Generalized sparse matrix primitives

In a technical report [33], we consider generalizations of the SEGMV algorithm. Two specific generalizations, block entry matrices and multiple right-hand sides, can be implemented in terms of sparse matrix-vector multiplication, but offer the potential for improved performance if implemented directly. We also consider the application of other segmented operations to sparse matrix computation and outline a column-oriented algorithm that is useful for symmetric matrices.

6.5.2 Support tree conjugate gradient

We also applied our sparse matrix multiplication algorithm to a new variant of the preconditioned conjugate gradient algorithm. The linear systems associated with large, sparse, symmetric, positive definite matrices are often solved iteratively using the preconditioned conjugate gradient method. Gremban developed a new class of preconditioners, called support tree precon-

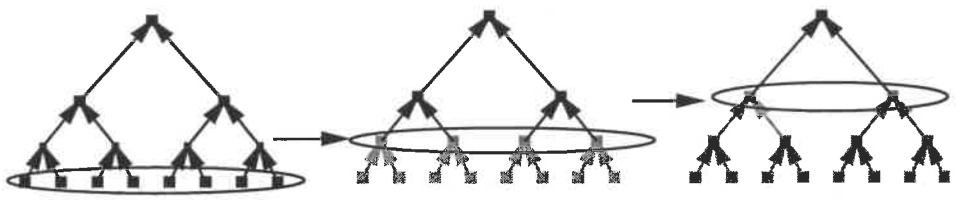


Figure 6.8: Leaf Raking in the Support Tree Conjugate Gradient algorithm. A lower triangular linear system corresponding to a tree directed from leaves to root is shown at the left. In the first parallel step, the solutions at the leaves are computed, and the right hand side values at the parents are updated. In succeeding parallel steps, the process is repeated at the leaves obtained when the previous set of leaves is removed. At the last step (not shown) the solution at the root is computed. Sparse matrix multiplication is used in each step of the algorithm.

ditioners [95], which are based on the connectivity of the graphs corresponding to the matrices and are well-structured for parallel implementation. In a previous study [94], we evaluated the performance of support tree preconditioners by comparing them against two common types of preconditioners: diagonal scaling, and incomplete Cholesky. Support tree preconditioners require less overall storage and less work per iteration than incomplete Cholesky preconditioners. In terms of total execution time on the CRAY C90, support tree preconditioners outperform both diagonal scaling and incomplete Cholesky preconditioners.

The performance of support tree conjugate gradient (STCG) is dominated by two operations: sparse matrix multiplication and preconditioning using a tree-based preconditioner. In fact, the summing operation performed at each level of the support tree is essentially a sparse matrix multiplication. (See Figure 6.8.) Thus, the bulk of the computation in STCG can be implemented with a single general-purpose sparse matrix multiplication subroutine. On the CRAY C90, our SEGMV-based implementation achieves high performance on both regular and irregular meshes, as described in the performance study [94].

Chapter 7

Graph Connectivity

This chapter considers the problem of finding the connected components of a graph, that is, obtaining a labeling of each node in an undirected graph with n nodes and m edges, such that all nodes in a connected component have the same label distinct from that of any other connected component.

Section 7.1 reviews previous theoretical and practical work on connected components. Section 7.2 uses a pointer-jumping algorithm to explain how contention is modeled and how contention can be reduced. We demonstrate that accounting for contention is important for graph algorithms, and show that the $(d, \mathbf{x})$-BSP is accurate for predicting performance. Section 7.3 describes a CRCW PRAM algorithm for labeling connected components adopted from a previous experimental study [92]. We describe several practical improvements and discuss trade-offs among several variants of the algorithm. Section 7.4 reports our performance results and compares performance results for a variety of architectures. We show that an efficient, general-purpose PRAM connected components algorithm can achieve consistently high performance across a wide variety of graphs. In particular, the implementation for a 16-processor CRAY C90 achieves a rate of over 100 million nodes per second.

7.1 Background and previous work

This section considers previous work on serial algorithms, PRAM algorithms, and implementations of graph connectivity algorithms. Efficient serial algorithms for finding connected components include depth first search, which runs in $O(n+m)$ time for n nodes and m edges, and union-find on a disjoint forest (with path compression) [192], which runs in $O(m\alpha(m,n)(m+n))$ time, a complexity that is effectively linear in the graph size. Efficient serial algorithms are difficult to parallelize because parallel versions of the algorithms would require simultaneous updates to shared pointer-based data structures.

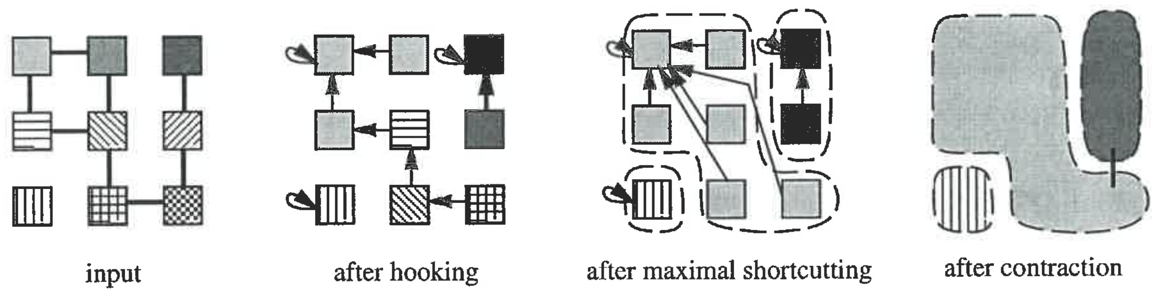


Figure 7.1: Steps in graph connectivity algorithms.

We present a brief overview of parallel algorithms for connected components (following the presentation by Greiner [92]) and highlight basic techniques that are used for connectivity and other similar graph algorithms (e.g., minimal spanning tree or network flows).

Graph connectivity algorithms are typically constructed from a number of basic primitives depicted in Figure 7.1:

1. Hooking: combining trees of vertices.

2. Shortcutting: reducing the length of pointer chains using pointer jumping.

3. Graph contraction (and expansion): combining multiple nodes into a single supernode.

The Shiloach-Vishkin [179] and Awerbuch-Shiloach [10] algorithms are similar, and both find the connected components of a graph by forming and combining (hooking) trees of vertices, maintaining the invariant that all vertices in a given tree belong to the same connected component. Initially, each vertex is in a separate tree, and the trees are combined to find the maximal trees subject to the invariant. The roots serve as representative elements of the trees, and the algorithms return the sequence of the roots corresponding to each vertex. Both algorithms require $O(\lg n)$ steps and $O(m \lg n)$ work. Thus, they are not work-efficient, since a serial depth-first search requires only $O(m)$ work. However, the algorithms are relatively simple to implement.

Another practical algorithm is a combination of Reif's random mating [167] (a randomized form of hooking) and Phillips' graph contraction [159]. With high probability, it also requires $O(\lg n)$ steps and $O(m \lg n)$ work. However, for some graphs, including planar graphs, only $O(m)$ work is needed. Much of the advantage of this algorithm over the previous algorithms is that it contracts both the edges and nodes of the graph. Edge contraction is easily incorporated into the previous algorithms, resulting in increased performance on some classes of graphs.

The Greiner-Blelloch hybrid algorithm [93] also blends the features of the previous algorithms. It combines graph contraction with shortcutting, which is used by the first two algorithms for flattening trees and improving the efficiency of repeatedly hooking. The hybrid algorithm hooks nodes together to form a forest, fully shortcuts each tree to a single node, contracts the graph to form a new graph that is processed recursively; and expands the graph to propagate the new labels. The main advantage of this algorithm is that its use of *maximal* shortcutting contracts most graphs very quickly. This algorithm performs $O(\lg^2 n)$ steps and $O((m+n)\lg n)$ work (possibly only $O(m+n\lg n)$ work for special classes of graphs [92]).

Gazit's optimal algorithm [81] runs in $O(\lg n)$ time and performs $O(m+n)$ work. The algorithm appears to be too complicated for developing a practical implementation, but it might be possible to incorporate aspects of the algorithm into other algorithms, in particular, the technique of processing only an $O(n)$ sized subset of edges in each iteration rather than the entire set of edges.

7.1.1 Previous implementations

This section briefly describes previous implementations of graph connectivity algorithms. (Performance comparisons are presented in Section 7.4.3.) Previous work on parallel graph connectivity implementations falls into three basic categories: implementations based on serial algorithms, algorithms redesigned to take advantage of locality, and direct implementations of fine-grained algorithms (such as the PRAM algorithms described earlier).

Early work on vector computers was based on serial algorithms and focused on specialized single-processor code without considering contention. For example, a CRAY Y-MP implementation by Evertz for regular grids [75] vectorizes serial breadth first search, serially looping both over levels of the graph and over the grid dimensions within a level. Mino implemented a vectorized version of serial union-find for grids [145].

Connected components algorithms have been implemented on several distributed memory machines. This work includes implementations for the Intel Paragon [77], the Connection Machines CM-2 and CM-5 [55, 40], a portable implementation for the Meiko CS-2, TMC CM-5, and IBM SP-2 [11], and a fast implementation for the Cray T3D [124, 139]. For distributed memory computers, performance on general graph algorithms is limited by communication. Thus, the approach typically used on distributed memory MIMD machines is to minimize global communication by separating the algorithm into local and global phases, where the local phase uses optimized sequential algorithms, and the global phase is optimized for communication using a model such as LOGP [62]. We discuss this approach further in Section 7.4.3.

Hsu et al. implemented a number of fine-grained graph algorithms for the MasPar MP-1, including connected components, minimum cost spanning forest, and ear decomposition [108, 109]. Their implementations are based on a library of primitives including list ranking, Euler tour construction, and graph renumbering. The generality of their approach is very appealing, but the performance on most algorithms is limited by the slow communication primitives of the

MasPar. In particular, speedup on most algorithms was limited to a factor of 5 over a Sparc II workstation.

Greiner compared the performance of several data-parallel PRAM algorithms for finding connected components, using implementations coded in the portable data-parallel language NESL and timed on the CM-2 and a single-processor CRAY C90 [92]. (NESL code for several algorithms including the hybrid algorithm is available in a technical report [91].) Greiner analyzed many pragmatic issues and concluded that the most practical algorithm of those studied is Blelloch's hybrid algorithm [93], an algorithm that combines features of the Shiloach-Vishkin [179] and Awerbuch-Shiloach [10] algorithms, Reif's random mating [167] algorithm, and Phillips' graph contraction [159]. This hybrid algorithm, described in Section 7.3, is the basis for our implementation study.

7.2 Modeling and reducing contention

Greiner's study used CRCW algorithms, which is not unreasonable for a single processor CRAY C90 (where the bank delay is only approximately 3 times the gap), or for a Connection Machine CM-2, which supports hardware combining. Nevertheless, in order to scale the implementation to multiple CRAY processors, we need to consider the impact of memory contention.

There are several potential sources of contention in graph algorithms. In the steps of a typical graph algorithm, the memory contention is proportional to the maximum degree of the graphs being manipulated. Note that some graph algorithms create and manipulate relatively dense graphs, even for sparse input graphs. For these algorithms, the memory contention can be arbitrarily high.

Module map contention is also a concern. The mapping of graph elements to memory banks is especially important for regular grids. For example, vision applications commonly use grid sizes that are powers of two and contain sub-images with regular patterns. Module map contention (and network contention) can be avoided by using randomization. A general technique is to use random mappings of nodes and edges to banks and to inject memory references from each processor in a random order. In practice, an inexpensive heuristic for reducing module map contention is to randomize the initial ordering and numbering of nodes and edges in the input graph.

7.2.1 Case study: maximal shortcutting

In order to illustrate the importance of memory contention in graph algorithms, we consider the problem of maximal shortcutting, a key substep of the connected components algorithm we describe later in Section 7.3. A maximal shortcutting operation transforms a forest of directed trees by shrinking each tree to a tree of height one (sometimes called a "star"). That is, each node's parent pointer is replaced with a pointer to the root of its tree. In general, the parallel complexity is a function of both the total number of nodes, n, and the height of the tallest tree in the forest, h. The serial complexity for maximal shortcutting is simply $O(n)$, which can be obtained with a recursive depth-first traversal that assigns each non-root node the label obtained from traversing the node's parent. Table 7.1 summarizes asymptotic running times for three maximal shortcutting algorithms: repeated pointer jumping, tree contraction, and a variant of pointer jumping that we refer to as label propagation.

Repeated pointer jumping performs a number of iterations of shortcutting in which the parent of each vertex v, Parent(v) is replaced with Parent(Parent(v)). This algorithm performs $O(\lg h)$ parallel steps and $O(n \lg h)$ work. It is easy to implement, but it performs $O(\lg h)$ more work than an optimal algorithm, and furthermore, has worst-case contention linear in the size of the largest tree.

Tree contraction [143, 165] is a general technique for evaluating expression trees. One advantage of using tree contraction is that work-efficient EREW tree contraction perform less work asymptotically than repeated shortcutting. In practice, however, the constant factors in the amount of work might offset the theoretical improvements due to several costs, including the costs of symmetry breaking, compacting active vertices, eliminating fewer than half of the vertices per iteration, and executing both contraction and expansion phases.

These costs lead us to consider variations on simple shortcutting to reduce contention. One simple candidate is to terminate shortcutting for vertices as they reach the root, but this does not improve the asymptotic complexity since the maximum number of pointers reaching the root at each step increases geometrically. However, a subtle variation does reduce the asymptotic amount of contention for several common cases. If roots are labeled before starting as *done*, and the *done* labels are propagated during shortcutting, the maximum contention is reduced. Figure 7.2 illustrates the difference between these two approaches. For most types of trees, the labeling algorithm keeps the contention low. The maximum contention is $O(1)$ for a linked list, $O(\lg n)$ for a binary tree, and more generally, $O((\lg n/\lg c)^{\lg c})$ for a c-ary tree.

However, label propagation does not solve contention problems in general. The worst case, which we call a "palm tree" graph, is a tall tree with high connectivity near the leaves. (See Figure 7.3.) Intuitively, the labels at the roots cannot propagate fast enough to prevent the contention building from the leaves. Note that the worst-case contention, even for bounded-degree graphs, is linear in the number of nodes.

It is straightforward to analyze the predicted performance of various shortcutting algorithms for special cases, such as binary trees. In order to verify our predictions and to understand more about contention in the general case, we performed a number of experiments on random

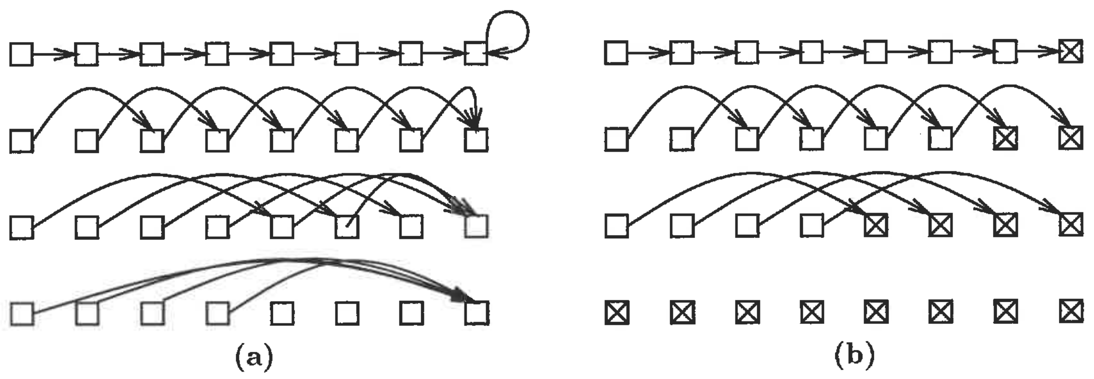


Figure 7.2: Two methods of avoiding contention in shortcutting: (a) quitting when reaching a root pointer, which reduces the worst-case contention from n to $n/2$, and (b) propagating *done* labels (indicated by an X), and quitting when reaching one, which makes no concurrent accesses for a linked list. Each row represents one iteration of shortcutting, and arrows indicate the memory accesses made during the iteration.

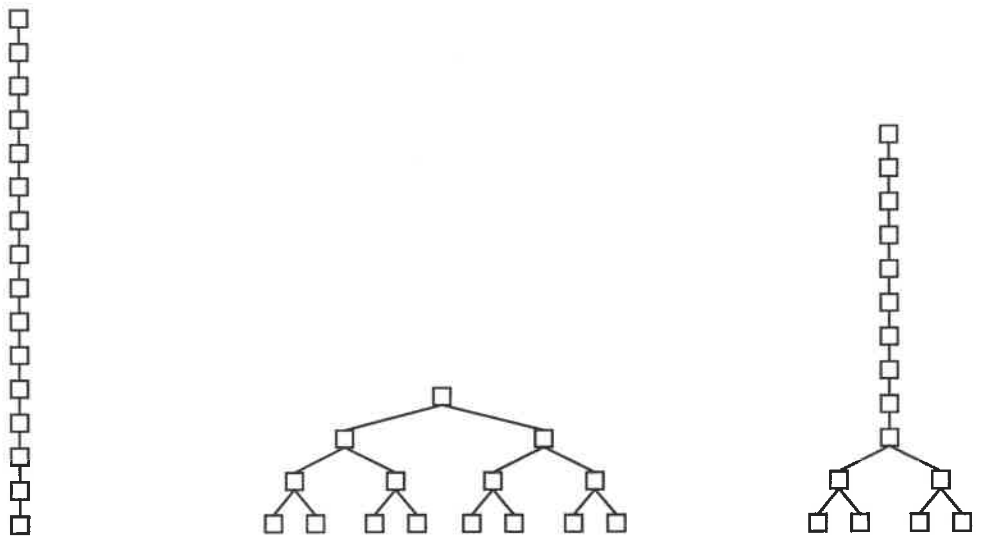


Figure 7.3: Different cases for contention on shortcutting: a list of height n, a binary tree of height $\lg n$, and a "palm tree" of height n.

		contention		
algorithm	work	list	binary tree	palm tree
pointer jumping	$n \lg h$	n	n	n
tree contraction	n	1	1	1
label propagation	$n \lg h$	1	$\lg n$	n

Table 7.1: Asymptotic upper bounds for memory contention when fully shortcutting a tree with n nodes.

forests. The random forests are generated by choosing c root nodes and hooking new nodes one at a time to a random node in the forest. Figure 7.4 shows a sample forest with five trees. In our experiments, each forest has one million nodes. We first look at the performance of simple shortcutting. Figure 7.5, which compares the predictions using the $(d, \mathbf{x})$-BSP with measured running times, shows that the predictions are accurate and that the contention indeed reaches the worst-case value. Figure 7.6 indicates that, for random trees, label propagation reduces the maximum memory contention by an order of magnitude. Label propagation is used in the connected components algorithm described in Section 7.3 and evaluated in Section 7.4.

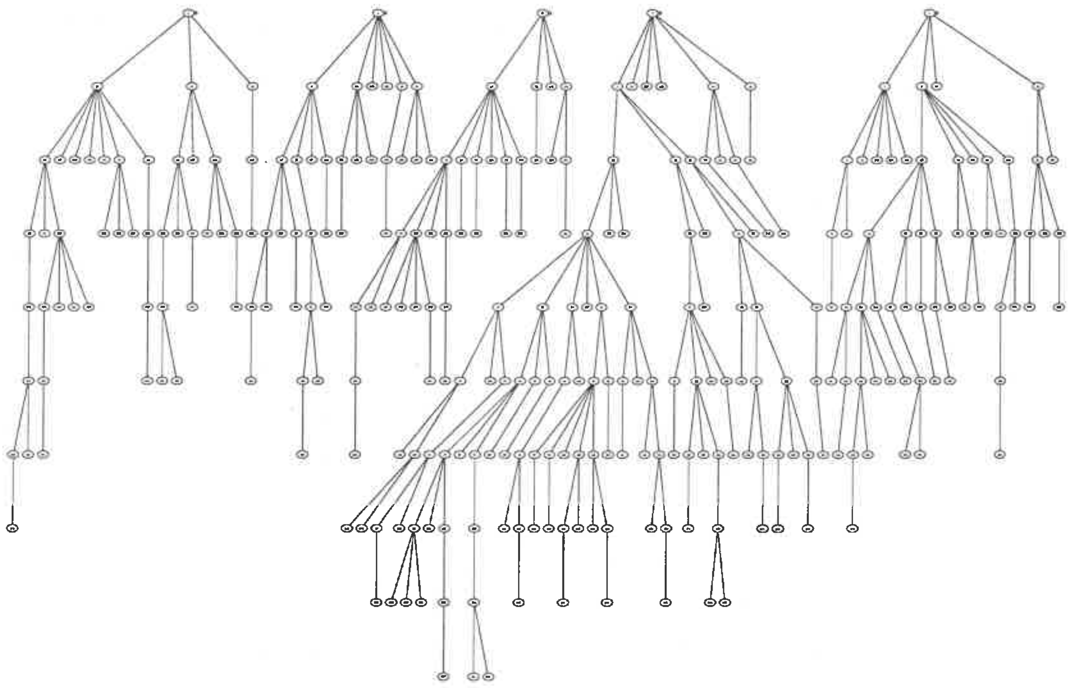


Figure 7.4: Sample test forest with 5 trees and 300 nodes.

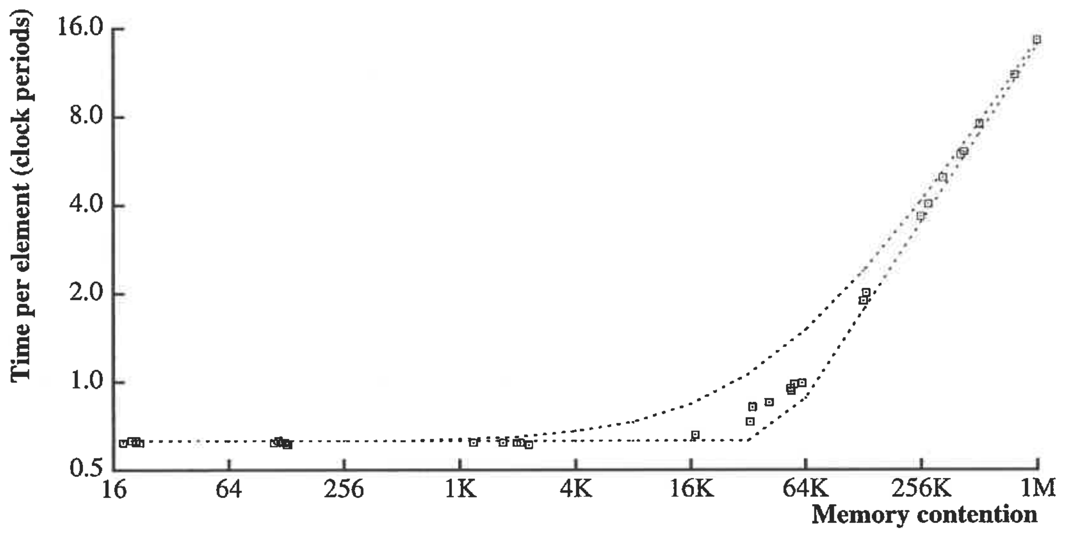


Figure 7.5: Running time on the CRAY J90 ($p = 4$) while contracting random trees using simple iterated pointer jumping. Points represent the time for each step up the shortcutting for a variety of random forests. The dotted curves show (d, x)-BSP predictions. The top curve represents the sum of the components for processor work and memory contention, whereas the bottom curve represents the maximum of these components.

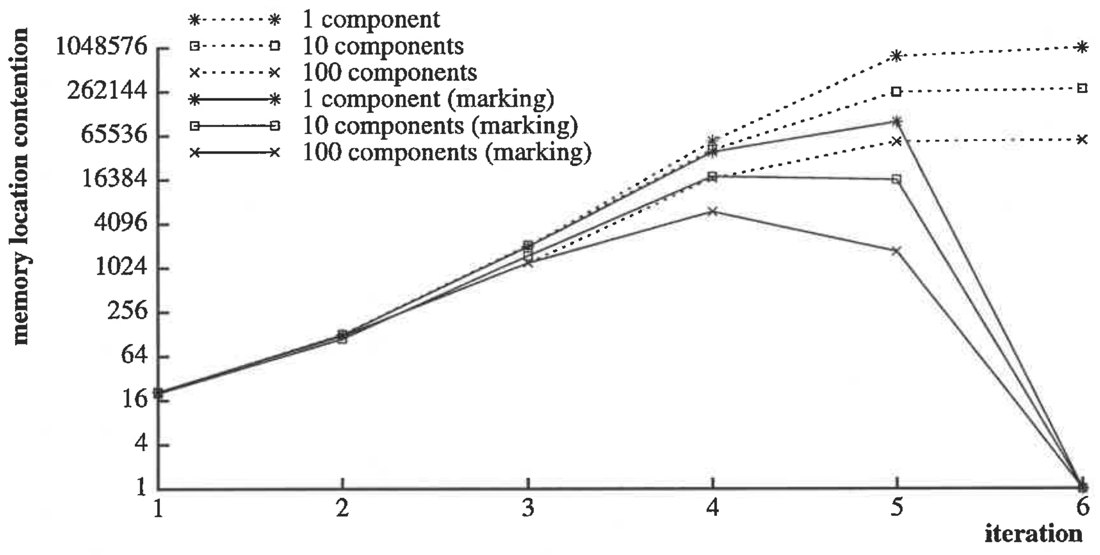


Figure 7.6: Memory contention after each shortcutting step, with and without label propagation for random forests. The dotted curves show the contention with simple iterated shortcutting and the solid curves show the contention with a variation that labels the roots, propagates the labels, and omits pointer-jumping for any node with a root label. The random forests, each with one million nodes, are generated by choosing c root nodes (for $c = 1$, 10, and 100) and hooking new nodes one at a time to a random node in the forest. In these test cases, label propagation cuts the maximum contention by a factor of 9–16.

7.3 Connected components hybrid algorithm

This section describes the connected components algorithm used in our implementation experiments. As a starting point for developing a highly-optimized implementation, we selected Blelloch's hybrid algorithm [93], which was the fastest algorithm in Greiner's comparison of practical connected components algorithms [92].

The algorithm initializes each node to point to itself, and then repeats the following until no edges remain:

1. flip edges (tail, head) from "larger" tail to "smaller" head (where larger and smaller are defined by an ordering function),
2. hook edges, writing the head label of each edge into the node pointed to by its tail,
3. maximally shortcut the graph into a collection of stars,
4. relabel edges, replacing the endpoints of each edge with the labels of the root of its star,
5. remove self-edges.

Flipping and hooking break symmetry and create a forest. In the case of multiple outgoing edges from a node, only the last node pointer written will remain in the forest. To ensure termination during maximal shortcutting, the forest must be an acyclic graph, which can be implemented by simply using the node number to orient edges. Maximal shortcutting can be implemented using pointer jumping, as discussed in Section 7.2.1. Self-edges can be removed by compacting remaining edges into a new array. At the termination of the algorithm, the nodes of each connected component have the label of one node in the component and labels in different components are distinct.

This algorithm executes $O(\lg n)$ iterations and each iteration performs $O(m + n \lg n)$ work, giving a total of $O(m \lg n + n \lg^2 n)$ work. The number of parallel steps is dominated by shortcutting, which performs $O(\lg^2 n)$ steps. Thus, the running time on CRCW PRAM with unit-time scans is $O((m \lg n + n \lg^2 n)/p + \lg^2 n)$.

For practical applications on large datasets, the number of steps in the hybrid algorithm is tolerable, but the benefits of parallelization could be offset by the additional work compared to an optimal serial algorithm. In addition, the CRCW hybrid algorithm can cause high contention. Even for bounded-degree graphs, if the graph is highly connected, there can be $O(n)$ contention for shortcutting and $O(m)$ contention for hooking and edge relabeling.

In order to reduce the amount of work and the amount of contention, we experimented with three improvements to the basic algorithm: contraction, randomization, and faster shortcutting.

7.3.1 Contraction

The first improvement we consider is the use of graph contraction [159], as used in the hybrid algorithm described by Greiner and Blelloch [92]. Rather than processing all the nodes throughout the algorithm, we contract the graph after each iteration, process the smaller graph recursively, and propagate the labels generated in the recursive call. Most steps of the algorithm remain the same, but after removing self-edges, we also remove inactive nodes. This modification requires a series of bookkeeping operations: renumbering the active nodes (using a scan operation), renumbering the head and tail of each active edge, and saving the parent of each compressed node. In our implementation, the graph is optimally load-balanced at each iteration.

Using graph contraction leads to savings in both work and contention. First, consider the work, and let v and e be the number of remaining nodes and edges, respectively, in an iteration. Let v_i be the number of remaining nodes in the ith connected component and let d_i be its diameter. The number of parallel steps for the iteration is $O(\max_i \lg d_i)$, and the work for the iteration is $O(e + \sum_i v_i \lg d_i)$. The complexity of the entire algorithm depends on how fast the number of edges and number of nodes decrease. The number of nodes decreases by at least a factor of two every other iteration [92], which reduces the work from $O(m \lg n + n \lg^2 n)$ to $O((m+n)\lg n)$. In the worst case, the number of edges decreases very slowly, but for some graphs, the number of edges decreases by a constant factor on each iteration [93]. For our experiments on grids, the number of edges and nodes decrease by a substantial constant factor each iteration, as discussed in Section 7.3.2. When both the number of nodes and number of edges contract by a constant factor each iteration, the work is reduced to $O(m + n \lg n)$. If, in addition, tree shortcutting is implemented in linear time, the complexity is further reduced to $O(m+n)$.

Another benefit of contraction, in addition to the savings in work, is a reduction in contention. Without contraction, the contention increases each iteration as the size of each tree grows. With contraction, each tree is reduced to a single supernode before starting the next iteration. Thus the contention is determined by the size of the trees formed in each iteration, rather than the size of the largest connected component.

7.3.2 Randomization

In Section 7.2.1, we showed that memory contention during maximal shortcutting can be reduced in some cases by propagating labels. A more general solution is to reduce the size of any tree created during the hooking step. This will reduce the amount of contention in maximal shortcutting and in node- and edge-renaming operations used in graph contraction. One way of reducing the size of trees is to randomize the graph, which we implement by randomly permuting the node numbers, relabeling the edges accordingly, and randomly reordering the edges.

Experimentally, the length of the longest chain for fully connected one- and two-dimensional grids appears to be approximately logarithmic in the number of nodes. If the longest chain in each iteration is in fact $O(\lg v)$, and if both the number of nodes and number of edges contract by a constant factor each iteration, then the work for finding connected components is reduced to $O(m + n \lg\lg n)$.

Note that we are striking a compromise between two extremes. Some algorithms create forests with trees of constant size, whereas the original hybrid algorithm creates forests potentially with very long chains. Randomization in combination with maximal shortcutting exploits the savings in work from maximal shortcutting, while avoiding the worst-case contention from long chains.

Randomization also reduces module map and network contention, which is particularly important for graphs based on regular grids. Random node numberings lead to well-balanced module mappings of nodes and edges to banks, and random reordering of edges breaks up regular patterns in the ordering of memory requests.

However, there are two minor drawbacks to randomization. First, a randomized node numbering can lead to a slight increase in work over a natural numbering for grid graphs. In our experiments, grid-based graphs contract by approximately a factor of three per iteration with randomization, compared to a factor of five per iteration without randomization. The total amount of work for grid-based graphs is typically 20 percent higher when using randomization. Second, the randomization itself can be somewhat expensive, depending on the application. In some applications, however, the graph can be maintained in random order, or the cost of randomization can be amortized over multiple uses of the graph.

7.3.3 Shortcutting improvements

Shortcutting a forest with simple pointer-jumping has two shortcomings: high memory contention, and a logarithmic factor more work than a serial algorithm. Memory contention can be reduced using label propagation (as described in Section 7.2.1) or with randomization (as described in Section 7.3.2). The work can also be reduced using improvements to the shortcutting algorithm. Reducing the work is particularly important when randomization is *not* used, since the size of the trees formed by hooking, in the worst case, is proportional to the size of the input graph.

One possibility is to compact the active nodes (i.e., those not pointing to a root) after each pointer-jumping step. Compaction does not help in theory on an individual tree, although it helps for certain types of forests. In particular, when there are $O(n)$ trees and the largest tree has $o(n)$ nodes, the amount of work is $O(n)$. Compaction can help in practice when the forest contains trees that are short or bushy near the root. Compacting nodes, however, is not a general solution.

We can exploit property of the hybrid connected components algorithm to design a general, work-efficient maximal shortcutting algorithm. In particular, we now describe an optimization for *ordered* links, which result from hooking higher numbered vertices to lowered numbered vertices. (The ordering is reversed in alternate iterations [92], but for the following discussion, assume hooking from higher to lower numbered vertices.) First, we observe that this ordering simplifies a serial implementation of maximal shortcutting when a tree of n nodes is represented as an array of n pointers, stored in an arbitrary order. Without ordered links, maximal shortcutting requires chasing pointers (e.g., with depth-first search) or performing multiple passes over the array of nodes. With ordered links, however, the tree can be traversed in a top-down ordering simply by processing the elements of the array sequentially. This permits maximal shortcutting with a single pass of in-place shortcutting.

To incorporate the advantages of this serial algorithm into a parallel algorithm, we conceptually divide the n nodes into p blocks of contiguous nodes, one block for each of p processors. In parallel, each processor performs one pass of in-place serial shortcutting over its block of elements. This single pass ensures that all pointers within a block point to root nodes, which implies that the longest chain of pointers has a length of at most p. Thus, maximal shortcutting can be finished with at most $\lceil \lg p \rceil$ iterations of pointer jumping.

If the data set is large relative to the number of virtual processors, $O(\lg p)$ iterations is a significant improvement over $O(\lg n)$ iterations, but the algorithm is not work-efficient. It can be improved to a work-efficient algorithm that performs only two passes over the nodes by adapting a technique from the partition method for solving linear recurrences [51, 79, 204]. As before, in the first pass, each processor performs in-place serial shortcutting over a contiguous block of elements. Then, in the second pass, instead of parallelizing across blocks, each block is processed in parallel. The blocks are processed one at a time starting from the first block (with an implicit synchronization before proceeding to the next block), performing a single in-place pointer jump on each node. This second pass effectively processes the trees top down, and fully shortcuts the forest because the first phase eliminates all dependences within a block.

This parallel algorithm performs $O(n)$ work and $O(n/p)$ parallel steps, and thus is work-efficient for $n = O(p^2)$. The algorithm is practical for a single vector processor, but does not have enough parallelism to use on a large number of vector processors. We conjecture that the algorithm can be generalized to one with more parallelism that runs in $O(n^{1/\delta})$ steps and performs $O(\delta n)$ work for $2 \leq \delta \leq \lg n$.

Our implementation of maximal shortcutting uses a heuristic approach. It first performs

Class	Average Degree	Separator Size	Diameter
2D: 40%, 60%	1.6, 2.4	$O(n^{1/2})$	$O(n^{1/2})$
3D: 20%, 40%	1.2, 2.4	$O(n^{2/3})$	$O(n^{1/3})$
AD3	3.0	$O(n)$	$O(\lg n)$

Table 7.2: Characteristics of classes of graphs used in the test suite.

two inplace shortcuts with label propagation. Loop raking is used during the shortcutting to implement the memory access for virtual processors, in order to take advantage of ordered links. Without graph randomization, the two inplace shortcuts typically fully shortcut over 75 percent of the nodes. With randomization, the two shortcutting steps typically fully shortcut over 99 percent of the nodes. After the two shortcutting steps, all of the active nodes (those without a root label) are compacted into a new array. The implementation then alternates between shortcutting the remaining active nodes and compacting an index of active nodes.

7.4 Evaluation

7.4.1 Test suite

The implementations were evaluated using a test suite adopted from previous studies of connection component implementations [92, 124]. The test suite contains grid-based graphs, which are commonly used in computer vision and computational physics, and random graphs, which represent the most general, and frequently worst, case. Specifically, the following classes of graphs are used:

- 2D: Subsets of two-dimensional toroidal grids. Each vertex has a subset of the four neighbors of such a grid.
- 3D: Subsets of three-dimensional toroidal grids. Each vertex has a subset of the six neighbors of such a grid.
- AD3: Average degree 3 graphs. Each node i picks c_i random neighbors where c_i is chosen uniformly from the range 0–3.

Subsets of grids are obtained by selecting to include each edge from the complete grid with a given probability. Table 7.4.1 summarizes the properties of the graphs used in the test suite.

7.4.2 Implementations

We present results for an efficient serial algorithm, a simple implementation of the hybrid algorithm, and an optimized version of the hybrid algorithm:

			Hybrid+				
graph	union	single-page	1	2	4	8	15
2D 40%	0.55	6.8	15.8	30.6	53.9	104.1	165.7
2D 60%	0.49	6.6	15.7	30.4	52.6	99.4	161.3
3D 20%	0.60	5.2	15.0	29.9	55.0	107.2	162.0
3D 40%	0.49	7.4	14.6	28.3	48.9	94.1	144.5
AD3	0.47	6.4	10.8	20.6	36.0	65.5	90.6

Table 7.3: Connected component performance on the CRAY C90. Performance is measured in millions of graph elements (nodes + edges) per second on a fixed graph size of four million graph elements. The Hybrid+ algorithm uses compression and label propagation.

union: a serial (scalar) implementation of union-find operations on a disjoint forest with path compression [192], coded in C.

single-page: a simple single vector processor implementation of hybrid algorithm that fits on one page of text (approximately 50 lines of compact C code). See Figure 7.7. (If Cray compilers had better support for parallelizing packs and recurrences, it would run fairly well on multiple processors.)

Hybrid+: a parallelized, vectorized, optimized implementation of the hybrid algorithm, including algorithmic optimizations to reduce contention and to reduce shortcutting time. For the Hybrid+ implementation, we converted the original NESL implementation to vectorizable Fortran and C. (A few routines for packing vectors were coded in assembly language because the Fortran compiler generated inefficient code.) Recursion in the NESL was simulated with a stack of vectors. On a single processor, the native Cray implementation runs 2–3 times faster than the original NESL implementation, primarily due to using the improved shortcutting algorithm described earlier, and secondarily due to reducing the number of memory operations, for example, by fusing loops that were separate in the NESL version, and by storing both *from* and *to* pointers of an edge in a single 64-bit word.

7.4.3 Performance results

Figure 7.8 shows the performance of variants of the hybrid algorithm. For the graphs tested, compression and label propagation always improve performance, and randomization usually reduces performance. However, the benefit of randomization is apparent in Figure 7.9, which compares the performance of hybrid variants on a set of highly connected 2D graphs. For these graphs, randomization prevents performance from degrading on graphs that are highly connected

```
static int shrink(int *parents, int *from, int *to, int m) {
  int j, f, t, pack=0;
#pragma ivdep
  for (j = 0; j < m; j++)
    if ((t = parents[to[j]]) != (f = parents[from[j]])) {
      to[pack] = t; from[pack++] = f;
    }
  return pack;
}
static int compress(int *rename, int *new, int *from, int *to, int n, int m) {
  int j, t, f, pack=0;
  for (j = 0; j < n; j++) if (new[j]==j) rename[pack++] = j;
  for (j = 0; j < pack; j++) new[rename[j]] = j;
#pragma ivdep
  for (j = 0; j < m; j++) {
    to[j] = new[to[j]]; from[j] = new[from[j]];
  }
  return pack;
}
static void shortcut_max(int *node, int n) {
  int j, done=0, old, new;
  while (done != n)
#pragma ivdep
    for (done=0, j = 0; j < n; j++) {
      old = node[j];
      new = node[old];
      node[j] = new;
      done += (new == old);
    }
}
void miniCC(int *nodes, int *map, int *index, int *from, int *to, int *ren,
             int n, int big_n, int m) {
  int j, t, f;
  for (j = 0; j < n; j++) index[j] = map[j] = j;
  while ((n > 0) && (m > 0)) {
    for (j = 0; j < m; j++)
      if ((f = from[j]) < (t = to[j])) {to[j] = f; from[j] = t;}
    for (j = 0; j < n; j++) nodes[j] = j;
    for (j = 0; j < m; j++) nodes[from[j]] = to[j];
    shortcut_max(nodes, n);
    m = shrink(nodes, from, to, m);
    for (j = 0; j < n; j++) map[index[j]] = index[nodes[j]];
    n = compress(ren, nodes, from, to, n, m);
#pragma ivdep
    for (j = 0; j < n; j++) index[j] = index[ren[j]];
  }
  shortcut_max(map, big_n);
}
```

Figure 7.7: Single-page hybrid algorithm implementation with vectorization directives.

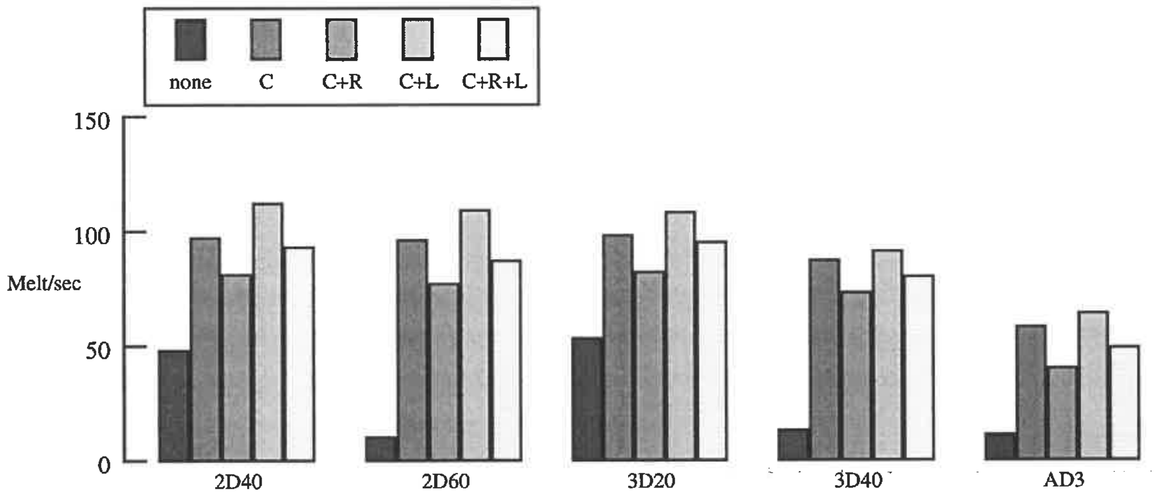


Figure 7.8: Performance effect of hybrid variants: compression (C), randomization (R), and label propagation (L). Measurements used 12 processors on the CRAY C90 (on a non-dedicated system) and the problem size is fixed at one million graph elements (nodes + edges).

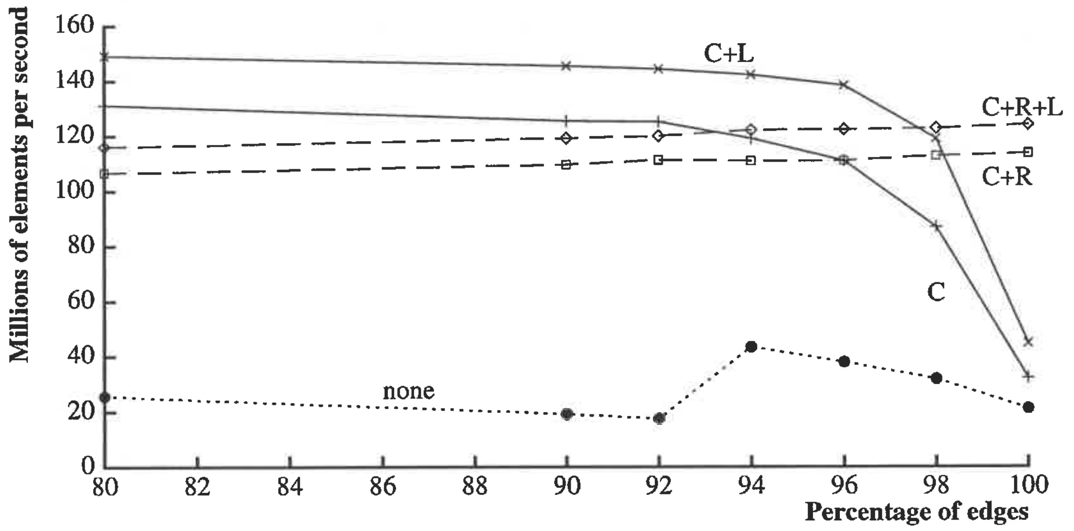


Figure 7.9: The effect of contention optimizations on the CRAY C90 for a set of highly connected 2D graphs. The percentage of edges selected from the complete 2D graph is varied from 80% to 100%. Performance is shown for implementations without contraction and for implementations with contraction(C), optionally with randomization (R) and root label propagation (L). The number of processors is fixed at 12 and the graph size is fixed at 4 million elements (nodes + edges).

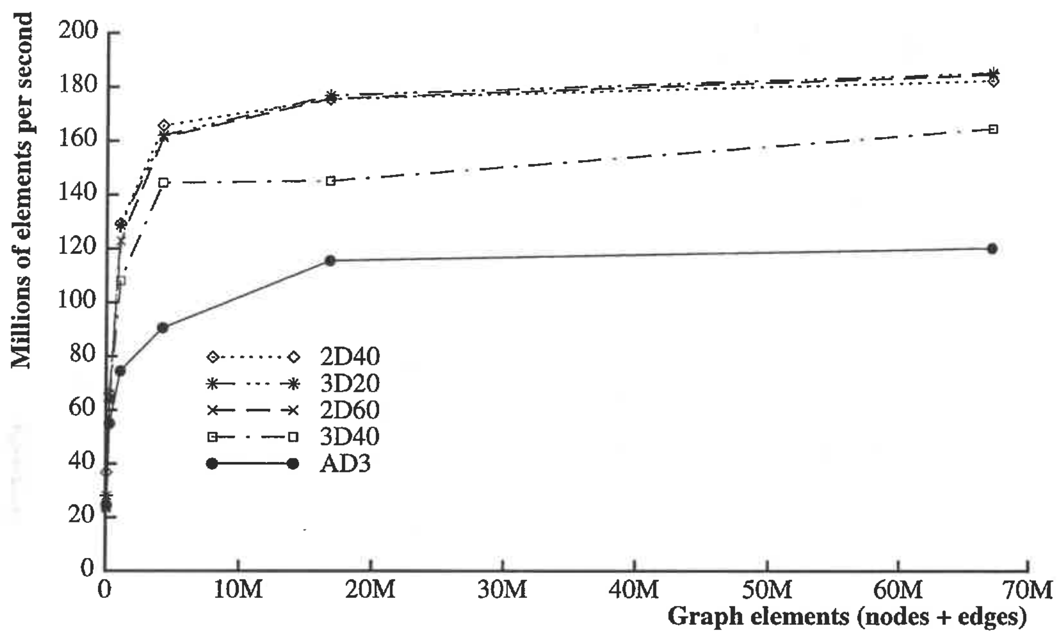


Figure 7.10: Hybrid algorithm performance on the test suite using 15 processors of the CRAY C90.

Figure 7.10 shows performance of the hybrid algorithm (with contraction and label propagation) on the CRAY C90 as a function of graph size. Table 7.3 lists performance results of serial algorithms and the hybrid algorithm for various numbers of processors. Note that the single-page implementation is an order of magnitude faster than the scalar union implementation and nearly half as fast as the optimized hybrid algorithm implementation on one processor.

All CRAY C90 measurements were performed at the Pittsburgh Supercomputing Center on a CRAY C90 that has only half the number of memory banks of a full configuration. Multiprocessor performance would be significantly better on a full configuration, as suggested by the measurements in Chapter 2.

7.4.4 Performance comparisons

Connected components algorithms have been implemented on a wide variety of machines, as discussed in Section 7.1.1. (Bader summarizes timings of many previous implementations [11].) We focus on two related studies using the Cray T3D [124, 139], which report the best performance to date. As indicated in Figure 7.11, the Cray T3D implementation exhibits higher peak performance than that of our CRAY C90 implementation. However, the Cray T3D implementation, and other locality-based implementations, achieve high performance by relying on particular properties of the input graph, and thus have several limitations.

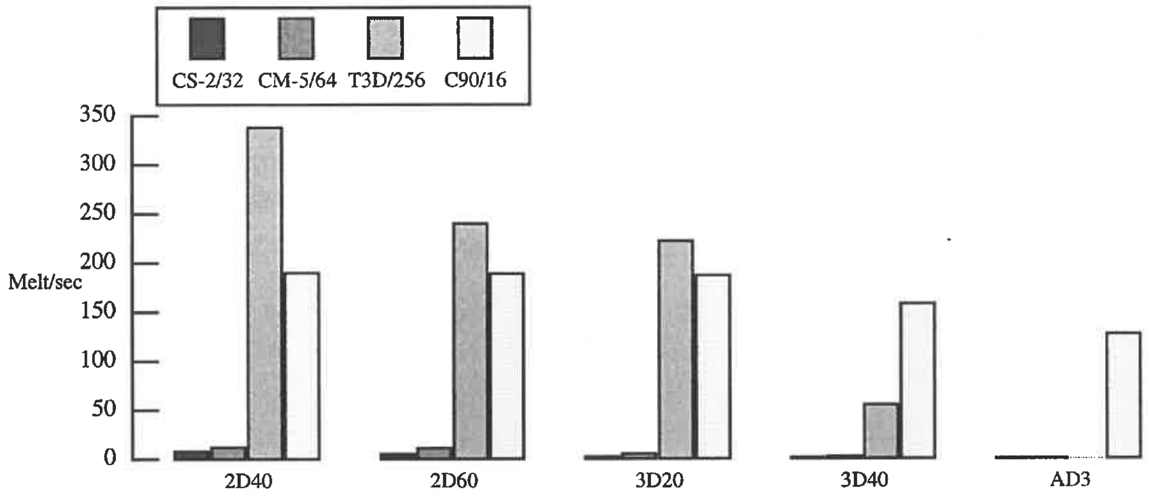


Figure 7.11: Performance comparison of implementations on the Meiko CS-2, TMC CM-5, Cray T3D, and Cray C90. Performance numbers from previous studies [11, 139] have been converted to millions of graph elements (nodes + edges) per second. T3D performance on the AD3 graphs is not known, but we expect it to be considerably lower than 3D40 because very little of the pointer chasing can be performed locally on a processor.

Performance of locality-based implementations depends on the amount of work that can be eliminated in an initial local phase using efficient serial algorithms. In the worst case—random graphs—there are very few edges between nodes on the same processor, and thus virtually all of the work requires global communication. For example, an implementation for the Cray T3D gets no speedup on multiple processors for random (AD3) graphs [124]. Even with grid-based graphs, performance is sensitive to the connectivity of the graph (as indicated in Figure 7.11) and to the size of the graph. For example, the performance on a 256-processor T3D flattens out at a speedup of approximately 30 for the 3D40 graphs, even when the input size is scaled with the number of processors [139].

In addition, the locality-based implementations require the input graph to be partitioned. In some cases, the input graph has a natural partitioning, and in other cases, a graph partitioning algorithm must be used. (This can present a bootstrapping problem when using a graph partitioning algorithm that finds connected components.) When incorporating a connected components algorithm into a complete application, the programmer must find a partitioning that accommodates the entire application in order to avoid additional global communication costs from reorganizing the graph.

In contrast, the CRAY C90 implementation processes random graphs at well over half of peak performance, processes highly connected grids at only slightly less the peak performance, achieves half of peak performance on graphs that have under one million nodes, and accepts nodes and edges in arbitrary order. In short, locality-based implementations have high peak performance on large grids with moderate connectivity, and are well suited for applications that generate only these types of graphs. The PRAM-based CRAY C90 implementation is much less sensitive to the properties of the input graph, significantly faster on random graphs, and better suited for use in a general-purpose subroutine library.

Chapter 8

Conclusions

This thesis considered the efficient implementation of irregular PRAM algorithms on machines with high-bandwidth, fine-grained shared-memory systems and support for latency-hiding. We described techniques for implementing and analyzing PRAM algorithms on high-bandwidth multiprocessors, presented a number of experiments on irregular algorithms, and introduced and experimentally evaluated an extension of the BSP model. We present conclusions in each of these areas.

Techniques for implementing PRAM algorithms on vector multiprocessors:

- The technique of simulating multiple virtual processors per physical processor is useful in practice for hiding latency on high-bandwidth shared-memory machines. In particular, we showed that the concept of virtual processors can be systematically used for vectorizing irregular PRAM algorithms.

- Pseudo-random mappings of memory locations to memory banks are useful for balancing the load among memory banks. When pseudo-random mappings of memory locations to memory banks are used, the QRQW model is adequate for making rough predictions of running times.

- Heuristics for reducing contention are useful in practice. In particular, reducing the amount of contention by a constant factor is useful, even though there is no benefit in asymptotic complexity; and heuristics for reducing randomization costs (e.g., randomly permuting selected data structures only once at the beginning of an algorithm) are useful even when these heuristics are not provably effective at randomization.

Algorithm experiments:

- Careful selection of a practical PRAM algorithm is important. The key factors are work-efficiency for large problem sizes, low constant factors, and the use of a low-contention model (e.g., the QRQW PRAM).

- Performance models for the algorithms based on the (d, x)-BSP are accurate, clarify trade-offs between algorithm and machine parameters, and guide the optimization process.

- The PRAM-like character of algorithms is retained in the implementation; optimizing the implementation centers on finding an appropriate mapping of the algorithm to the (d, x)-BSP rather than redesigning the algorithm.

- The implementations for the CRAY J90 and CRAY C90 are predictable and perform well on a wide range of data sets. For several irregular problems, our approach to programming high-bandwidth vector multiprocessors using PRAM algorithms has lead to the fastest implementations on vector machines, and the fastest general-purpose implementations for any machine.

Modeling memory bank contention and delay:

- Modeling memory bank delay is important in analyzing the performance of algorithms with high contention.

- The (d, x)-BSP reasonably explains the memory performance of the CRAY C90 and J90 on irregular access patterns, despite abstracting many machine-specific details.

- When using random mappings, additional memory bank expansion is useful, even beyond the point where the total bandwidth of the memory banks matches the total bandwidth at the processors.

- With low memory contention, high parallel slackness, and random mapping of memory locations to banks, the effect of the bank delay can typically be ignored.

8.1 Future work

There are several extensions of the thesis that could be explored.

Experiments on other architectures: It would be valuable to reimplement the modeling experiments and algorithm experiments studied in this thesis on other pipelined-memory machines, such as the TERA MTA [6], the Cray T3E [175], and vector microprocessors [131, 9, 205]. There is some evidence that the general trends identified in the (d, x)-BSP model would apply to other machines (including some preliminary contention experiments at TERA), but many questions remain about the specifics. Is module map contention noticeable on a machine like the TERA? In current systems with prefetch instructions, how well are the details of the architecture (e.g., multilevel caches, TLBs) hidden by prefetching?

Extensions to the Deluxe BSP: One extension of the (d, x)-BSP model motivated by recent technology trends would be the addition of caches to the memory banks, as available in the design of the TERA MTA [6], and as suggested by Hsu and Smith [110]. How much cache is needed at each memory bank to reduce the effects of module map contention?

Low-contention algorithms: For the algorithms considered in this thesis, we have found that the QRQW model leads to algorithms that are simpler and more efficient in practice than existing EREW and CRCW algorithms. It would be interesting to continue exploring the practical benefits of QRQW algorithms by developing improvements for algorithms explored in this thesis (in particular, for connected components), and by considering a wider class of applications.

Programming interfaces for PRAM algorithms: In addition to experimenting with the modeling and implementation of algorithms, a valuable area of further research is to experiment with different languages and systems for expressing algorithms. One could implement a BSP library [90, 144] for vector multiprocessors and compare performance results, both qualitatively and quantitatively, with those obtained on other architectures. Another direction would be to incorporate a cost model for contention in a higher level model, such as the NESL language and cost-model, or a BSP library [90, 144]. In addition, profiling tools could be supplied for measuring contention (e.g., in the NESL profiling system [32]) in order to aid in the process of designing low-contention algorithms.

Analysis of NUMA architectures, for fun and profit: I am currently studying performance characteristics and performance tools for scalable microprocessor-based shared-memory multiprocessors, including the Silicon Graphics Origin 2000 [130].

Bibliography

[1] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, Sept. 1992.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[3] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Proceedings Supercomputing '92*, pages 32–41, Nov. 1992.

[4] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press, Toronto, 1985.

[5] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *International Conference on Supercomputing*, pages 188–197, July 1992.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[7] E. Anderson and Y. Saad. Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing*, 1(1):73–95, 1989.

[8] A. W. Appel and A. Bendiksen. Vectorized garbage collection. *Journal of Supercomputing*, 3(3):151–160, Sept. 1989.

[9] K. Asanovic, B. Kingsbury, B. Irissou, J. Beck, and J. Wawrzynek. T0: A single-chip vector microprocessor with reconfigurable pipelines. In *Proceedings 22nd European Solid-State Circuits Conference*, pages 344–347, Sept. 1996.

[10] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *Proceedings of the International Conference on Parallel Processing*, pages 175–179, 1983.

[11] D. A. Bader and J. JáJá. Parallel algorithms for image histogramming and connected components with an experimental study (extended abstract). In *Proceedings 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 123–133, Santa Barbara, CA, July 1995.

[12] D. A. Bader and J. JàJà. Practical parallel algorithms for dynamic data redistribution, median finding, and selection. In *Proc. 10th Int. Parallel Processing Symp.*, pages 292–301, Apr. 1996.

[13] D. H. Bailey. Vector computer memory bank contention. *IEEE Transactions on Computers*, C-36:293–298, Mar. 1987.

[14] D. H. Bailey. A high-performance FFT algorithm for vector supercomputers. *International Journal of Supercomputer Applications*, 2(1):82–87, Spring 1988.

[15] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[16] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results 10-94. Technical Report NAS-94-001, NASA Ames Research Center, Oct. 1994.

[17] F. Baskett and A. J. Smith. Interference in multiprocessor computer systems with interleaved memory. *Communications of the ACM*, 19(6):327–334, June 1976.

[18] A. Bataineh and F. Özgüner. Parallel-and-vector implementation of the event-driven logic simulation algorithm on the Cray Y-MP supercomputer. In *Proceedings Supercomputing '92*, pages 444–452, Nov. 1992.

[19] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[20] R. H. Bisseling. Sparse matrix computations on bulk synchronous parallel computers. *Zeitschrift für Angewandte Mathematik und Mechanik (Special issue Proceedings International Conference on Industrial and Applied Mathematics)*, pages 127–130, 1996.

[21] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures (short version). In B. Pehrson and I. Simon, editors, *Proc. 13th IFIP World Computer Congress. Volume 1*, pages 509–514. Elsevier, 1994.

[22] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, Nov. 1989.

[23] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[24] G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, Apr. 1993.

[25] G. E. Blelloch. Prefix sums and their applications. In J. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[26] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[27] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[28] G. E. Blelloch, S. Chatterjee, F. Knabe, J. Sipelstein, and M. Zagha. VCODE reference manual (version 1.1). Technical Report CMU-CS-90-146, School of Computer Science, Carnegie Mellon University, July 1990.

[29] G. E. Blelloch, S. Chatterjee, and M. Zagha. Solving linear recurrences with loop raking. *Journal of Parallel and Distributed Computing*, 25(1):91–97, Feb. 1995.

[30] G. E. Blelloch, L. Dagum, S. J. Smith, K. Thearling, and M. Zagha. An evaluation of sorting as a supercomputer benchmark. Technical Report RNR-93-002, NASA Ames Research Center, Jan. 1993.

[31] G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):943–958, Sept. 1997. Preliminary version appears in *Proc. Symp. on Parallel Algorithms and Architectures*, pages 84–94, July 1995.

[32] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, and M. Zagha. NESL user's manual (for NESL version 3.1). Technical Report CMU-CS-95-169, School of Computer Science, Carnegie Mellon University, July 1995.

[33] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, Aug. 1993.

[34] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31:135–167, 1998. Preliminary version appears in *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[35] G. E. Blelloch, B. M. Maggs, and G. L. Miller. The hidden cost of low bandwidth communication. In U. Vishkin, editor, *Developing a Computer Science Agenda for High-Performance Computing*, pages 22–25. ACM press, 1994.

[36] G. E. Blelloch and C. R. Rosenberg. Network learning on the Connection Machine. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 323–326, Aug. 1987.

[37] B. Boothe and A. Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings International Symposium on Computer Architecture*, pages 214–223, May 1992.

[38] F. A. Briggs and E. S. Davidson. Organization of semiconductor memories for parallel pipelined processors. *IEEE Transactions on Computers*, C-26:162–169, Feb. 1977.

[39] H. K. Brock, B. J. Brooks, and F. Sullivan. Diamond: A sorting method for vector machines. *BIT*, 21:142–152, 1981.

[40] R. C. Brower, P. Tamayo, and B. York. A parallel multigrid algorithm for percolation clusters. *Journal of Statistical Physics*, 63:73–88, 1991.

[41] I. Y. Bucher and M. L. Simmons. Measurement of memory access contentions in multiple vector processors systems. In *Proceedings Supercomputing'91*, pages 806–817, Nov. 1991.

[42] P. J. Burns, M. Christon, R. Schweitzer, O. M. Lubeck, H. J. Wasserman, M. L. Simmons, and D. V. Pryor. Vectorization of Monte Carlo particle transport: An architectural study using the LANL benchmark GAMTEB. In *Proceedings Supercomputing'89*, pages 10–19, 1989.

[43] D. A. Calahan. Some results in memory conflict analysis. In *Proceedings Supercomputing '89*, pages 775–778, Nov. 1989.

[44] R. Calinescu. Conservative discrete-event simulations on bulk synchronous parallel architectures. Technical Report TR-16-95, Oxford University Computing Laboratory, Apr. 1995.

[45] P. Carnevali. Timing results of some internal sorting algorithms on the IBM 3090. *Parallel Computing*, 6(1988):115–117, 1988.

[46] L. J. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18:143–154, 1979.

[47] D. Y. Chang, D. J. Kuck, and D. H. Lawrie. On the effective bandwidth of parallel memories. *IEEE Transactions on Computers*, C-26:480–489, May 1977.

[48] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, Nov. 1990.

[49] T. Cheatham, A. Fahmy, D. C. Stefanescu, and L. G. Valiant. Bulk synchronous parallel computing—a paradigm for transportable software. In *Proc. IEEE 28th Hawaii Int. Conf. on System Science*, pages 268–275, Jan. 1995.

[50] L. T. Chen, L. S. Davis, and C. P. Kruskal. Efficient parallel processing of image contours. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(1):69–81, Jan. 1993.

[51] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical parallel band triangular system solvers. *ACM Transactions on Mathematical Software*, 4(3):270–277, Sept. 1978.

[52] T. Cheung and J. E. Smith. A simulation study of the CRAY X-MP memory system. *IEEE Transactions on Computers*, C-35(7):613–622, July 1986.

[53] C.-P. Chung and W.-Y. Lin. Vectorization of sorting algorithms. *International Journal of High Speed Computing*, 4(3):213–232, 1992.

[54] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings ACM Symposium on Theory of Computing*, pages 206–219, 1986.

[55] N. Copty, S. Ranka, G. Fox, and R. V. Shankar. A data parallel algorithm for solving the region growing problem on the Connection Machine. *Journal of Parallel and Distributed Computing*, 21(1):160–168, Apr. 1994.

[56] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press and McGraw-Hill, 1990.

[57] Cray Research Inc., Mendota Heights, Minnesota. *ORDERS(3SCI) Manual Page SR-2081 5.1*, Mar. 1988.

[58] Cray Research Inc., Mendota Heights, Minnesota. *Symbolic Machine Instructions Reference Manual, SR-0085B*, Mar. 1988.

[59] Cray Research Inc., Mendota Heights, Minnesota. *CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP Multitasking Programmer's Manual*, July 1989.

[60] F. Crow, G. Demos, J. Hardy, J. Mclaughlin, and K. Sims. 3D image synthesis on the Connection Machine. *International Journal of High Speed Computing*, 1:329–347, June 1989.

[61] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings Supercomputing'93*, pages 262–273, 1993.

[62] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 1–12, May 1993.

[63] D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast parallel sorting under LogP: from theory to practice. In *Proc. Workshop on Portability and Performance for Parallel Processing*, Southhampton, England, July 1993.

[64] L. Dagum. Sorting for particle flow simulation on the Connection Machine. Technical Report RNR-90-017, NASA Ames Research Center, Oct. 1990.

[65] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Kokhar. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33, 1995.

[66] U. Detert and G. Hofemann. CRAY X-MP and Y-MP memory performance. *Parallel Computing*, 17:579–590, 1991.

[67] R. Diekmann, J. Gehring, R. Lüling, B. Monien, M. Nübel, and R. Wanka. Sorting large data sets on a massively parallel system. In *Proc. 6th IEEE-SPDP*, pages 2–9, 1994.

[68] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proc. 19th Int. Colloquium on Automata Languages and Programming, Springer LNCS 623*, pages 235–246, July 1992.

[69] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. Technical Report Research Report 513, Universitat Dortmund, Dec. 1993.

[70] M. Dow. Sorting on vector processors—the binary radix sort. *Supercomputer*, 9(1):18–26, Jan. 1992.

[71] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Monographs on Numerical Analysis. Oxford University Press, New York, 1986.

[72] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, Mar. 1989.

[73] C. Engelmann and J. Keller. Simulation-based comparison of hash functions for emulated shared memory. In *Proc. Parallel architectures and languages Europe, Springer LNCS 694*, pages 1–11, June 1993.

[74] J. Erhel. Sparse matrix multiplication on vector computers. *International Journal of High Speed Computing*, 2(2):101–116, 1990.

[75] H. G. Evertz. Vectorized cluster search. *Nuclear Physics B (Proc. Suppl.)*, 26:620–622, 1992.

[76] P. Fernandes and P. Girdinio. A new storage scheme for an efficient implementation of the sparse matrix-vector product. *Parallel Computing*, 12:327–333, 1989.

[77] S. J. Fink, S. B. Baden, and K. Jansen. Cluster identification on a distributed memory multiprocessor. In *Proceedings Scalable High-Performance Computing Conference*, pages 239–246, May 1994.

[78] R. S. Francis and I. D. Mathieson. A benchmark parallel sort for shared memory multiprocessors. *IEEE Transactions on Computers*, 37(12):1619–1626, 1988.

[79] D. D. Gajski. An algorithm for solving linear recurrence systems on parallel and pipeline machines. *IEEE Transactions on Computers*, C-3(3):190–206, Mar. 1981.

[80] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings USENIX Annual Technical Conference*, June 1998. To appear.

[81] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, Dec. 1991.

[82] A. Gerbessiotis and C. Siniolakis. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of EUROPAR'96*, August 1996.

[83] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 223–232, June 1996.

[84] A. V. Gerbessiotis and C. J. Siniolakis. A randomized sorting algorithm on the BSP model. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997. IEEE.

[85] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. Technical Report TR-10-92, Harvard University, Computer Science Department, 1992.

[86] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, Sept. 1994. To appear in *SIAM Journal on Computing.*

[87] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. *Journal of Computer and System Sciences*, 53(3):417–442, 1996.

[88] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 72–83, June 1997.

[89] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 1–12, June 1996.

[90] M. W. Goudreau, S. B. Rao, and T. Tsantilas. A study of the BSP model: Algorithms and implementation. Technical Report UCFCS:CS-TR-94-04, Department of Computer Science, University of Central Florida, April 1994.

[91] J. Greiner. A comparison of data-parallel algorithms for connected components. Technical Report CMU-CS-93-191, School of Computer Science, Carnegie Mellon University, Aug. 1993.

[92] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 16–25, Cape May, NJ, June 1994.

[93] J. Greiner and G. E. Blelloch. Connected components algorithms. In G. W. Sabot, editor, *High Performance Computing: Problem Solving with Parallel and Vector Architectures.* Addison Wesley, Reading, MA, 1995.

[94] K. Gremban, G. Miller, and M. Zagha. Performance evaluation of a new parallel preconditioner. In *1995 International Parallel Processing Symposium*, pages 65–69, Apr. 1995.

[95] K. D. Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, Oct. 1996.

[96] D. T. Harper III. Block, multistride vector, and FFT accesses in parallel memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 2:43–51, Jan. 1991.

[97] D. T. Harper III and Y. Costa. Analytical estimation of vector access performance in parallel memory architectures. *IEEE Transactions on Computers*, 42(5):616–624, May 1993.

[98] D. R. Helman, D. A. Bader, and J. JàJà. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 211–222, June 1996.

[99] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo CA, 1990.

[100] M. A. Heroux. A proposal for a sparse BLAS toolkit. Technical Report TR/PA/92/90, CERFACS, Dec. 1992.

[101] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[102] High Performance Fortran Forum. *High Performance Fortran Language Specification version 2.0*, June 1997.

[103] W. L. Hightower, J. F. Prins, and J. H. Reif. Implementation of randomized sorting on large parallel machines. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 158–167, 1992.

[104] H. Hiraishi, K. Hamaguchi, H. Ochi, and S. Yajima. Vectorized symbolic model checking of computation tree logic for sequential machine verification. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification*, pages 214–224. Springer-Verlag, July 1991.

[105] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[106] C. A. R. Hoare. Quicksort. *Computer J.*, 5(1):10–15, 1962.

[107] R. W. Hockney and C. R. Jesshope. *Parallel Computers: Architecture, Programming, and Algorithms.* Adam Hilger Ltd., Bristol, England, second edition, 1988.

[108] T. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on the MasPar. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 15*, pages 165–198. American Mathematical Society, 1994. Also available as TR-92-38, Dept. of Comp. Sci., University of Texas at Austin, February 1992.

[109] T. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. In *Proc. 9th International Parallel Processing Symposium*, pages 106–112, Apr. 1995.

[110] W.-C. Hsu and J. E. Smith. Performance of cached DRAM organizations in vector supercomputers. In *Proc. 20th International Symp. on Computer Architecture*, pages 327–336, San Diego, CA, May 1993.

[111] R. M. Hyatt and H. L. Nelson. Chess and supercomputers: details about optimizing Cray Blitz. In *Proceedings Supercomputing '90*, pages 354–363, Nov. 1990.

[112] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, Reading, Mass., 1992.

[113] T. Johnson. Vlist: A vectorized list. Technical Report 92-007, Dept. of CIS, University of Florida, Apr. 1992.

[114] S. L. Johnsson. Combining parallel and sequential sorting on a Boolean n-cube. In *Proceedings International Conference on Parallel Processing*, pages 444–448, Aug. 1984.

[115] Y. Kanada. A vectorization technique of hashing and its application to several sorting algorithms. In *Parbase-90*, pages 147–151, Mar. 1990.

[116] A. R. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, 1988.

[117] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A*, pages 869–941. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[118] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, Department of Computer Science, University of Minnesota, 1995.

[119] D. R. Kincaid and T. C. Oppe. Recent vectorization and parallelization of ITPACKV. Technical report, Center for Numerical Analysis, The University of Texas at Austin, Nov. 1989.

[120] D. R. Kincaid, T. C. Oppe, J. R. Respess, and D. M. Young. ITPACKV 2C user's guide. Technical Report CNA-191, University of Texas at Austin, Center for Numerical Analysis, Nov. 1984.

[121] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, 1973.

[122] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, Aug. 1973.

[123] V. Kotlyar, K. Pingali, and P. V. Stodghill. Compiling parallel code for sparse matrix applications. In *Proceedings Supercomputing '97*, Nov. 1997.

[124] A. Krishnamurthy, S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In S. Bhatt, editor, *Parallel Algorithms: Third DIMACS Implementation Challenge.* Vol. 30 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 1–21, 1997.

[125] A. Krishnamurthy, K. E. Schauser, C. J. Scheiman, R. Wang, D. E. Culler, and K. Yelick. Evaluation of architectural support for global address-based communication in large-scale parallel machines. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, Oct. 1996.

[126] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. In *Proceedings International Conference on Parallel Processing*, pages 180–185, Aug. 1985.

[127] D. J. Kuck, P. Budnik, S. C. Chen, E. Davis, J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towle. Measurements of parallelism in ordinary FORTRAN programs. *IEEE Computer*, 7(1):37–46, Jan. 1974.

[128] J. T. Kuehn and B. J. Smith. The Horizon supercomputing system: Architecture and software. In *Proceedings Supercomputing'88*, pages 28–34, Nov. 1988.

[129] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, Oct. 1980.

[130] J. Laudon and D. Lenoski. The SGI Origin 2000: A CC-NUMA highly scalable server. In *Proceedings 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[131] C. G. Lee and D. J. DeVries. Initial results on the performance and cost of vector microprocessors. In *Proceedings 30th Annual International Symposium on Microarchitecture*, pages 171–182, Dec. 1997.

[132] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[133] C. Leiserson, Z. S. Abuhamdeh, D. Douglas, C. R. Feynmann, M. Ganmukhi, J. Hill, W. D. Hillis, B. Kuszmaul, M. S. Pierre, D. Wells, M. Wong, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 272–285, San Diego, CA, July 1992.

[134] D. E. Lenoski and W.-D. Weber. *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann, 1995.

[135] S. A. Levin. A fully vectorized quicksort. *Parallel Computing*, 16:369–373, 1990.

[136] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular fortran programs. In *LCR98: Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 1998.

[137] R. Löhner and J. Ambrosiano. A vectorized particle tracer for unstructured grids. *Journal of Computational Physics*, 91:22–31, 1990.

[138] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings International Conference on Parallel Processing*, pages 303–310, Aug. 1988.

[139] S. S. Lumetta, A. Krishnamurthy, and D. E. Culler. Towards modeling the performance of a fast connected components algorithm on parallel machines. In *Proceedings Supercomputing '95*, Nov. 1995.

[140] J. Makino. Vectorization of a treecode. *Journal of Computational Physics*, 87:148–160, 1990.

[141] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inf.*, 21:339–374, 1984.

[142] R. Melhem. Parallel solution of linear systems with striped sparse matrices. *Parallel Computing*, 6:165–184, 1988.

[143] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings Symposium on Foundations of Computer Science*, pages 478–489, Oct. 1985.

[144] R. Miller. A library for bulk-synchronous parallel programming. In *Proceedings British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, pages 100–108, Dec. 1993.

[145] H. Mino. A vectorized algorithm for cluster formation in the Swendsen-Wang dynamics. *Computer Physics Communications*, 66:25–30, 1991.

[146] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP.* Benjamin Cummings Publishing Company, 1990.

[147] J. Moscinski, Z. A. Rycerz, and P. W. M. Jacobs. Timing results of some internal sorting algorithms on the ETA 10-P. *Parallel Computing*, 11:117–119, 1989.

[148] L. Natvig. Logarithmic time cost optimal parallel sorting is *not yet* fast in practice! In *Proceedings Supercomputing '90*, pages 486–494, Nov. 1990.

[149] M. Nibhanupudi, C. Norton, and B. Szymanski. Plasma simulation on networks of workstations using the bulk synchronous parallel model. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 13–22, Athens, GA, November 1995.

[150] S. S. Nielsen and S. A. Zenios. Data structures for network algorithms on massively parallel architectures. *Parallel Computing*, 18(9):1033–1052, Sept. 1992.

[151] W. Oed. Cray Y-MP C90: system features and early benchmark results. *Parallel Computing*, 18(8):947–954, Aug. 1992.

[152] W. Oed and O. Lange. On the effective bandwidth of interleaved memories in vector processor systems. *IEEE Transactions on Computers*, pages 949–957, Oct. 1985.

[153] T. C. Oppe, W. D. Joubert, and D. R. Kincaid. NSPCG user's guide. Technical report, Center for Numerical Analysis, The University of Texas at Austin, Dec. 1988.

[154] R. Overill. A family of fully-vectorizable sorting algorithms. *Supercomputer*, 40(VII-6), Nov. 1990.

[155] R. Overill. Bitonic sorting for vector processors. *Supercomputer*, 9(2):4–8, Mar. 1992.

[156] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.

[157] G. V. Paolini and G. R. D. Brozolo. Data structures to vectorize CG algorithms for general sparsity patterns. *BIT*, 29:703–718, 1989.

[158] A. Peters. Sparse matrix vector multiplication techniques on the IBM 3090 VF. *Parallel Computing*, 17:1409–1424, 1991.

[159] C. A. Phillips. Parallel graph contraction. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 148–157, June 1989.

[160] R. Ponnusamy, Y.-S. Hwang, J. Saltz, A. Choudhary, and G. Fox. Supporting irregular distributions in FORTRAN 90D/HPF compilers. Technical Report CS-TR-3268 and UMIACS-TR-94-57, University of Maryland, May 1994. A revised version appears in IEEE Parallel & Distributed Technology, 3(1):12–24, Spring 1995.

[161] R. Raghavan and J. P. Hayes. On randomly interleaved memories. In *Proceedings Supercomputing '90*, pages 49–58, Nov. 1990.

[162] A. G. Ranade. How to emulate shared memory. *J. Comput. Syst. Sci.*, 42:307–326, 1991.

[163] B. Rau. Pseudo-randomly interleaved memory. In *Proceedings Int. Symp. Computer Architecture*, pages 74–83, 1991.

[164] M. Reid-Miller. List ranking and list scan on the Cray C-90. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 104–113, Cape May, NJ, June 1994.

[165] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 115–194. Morgan Kaufmann, 1993.

[166] J. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[167] J. H. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR-08-85, Harvard University, Mar. 1985.

[168] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, 1987.

[169] D. Richards. Parallel sorting—a bibliography. *ACM SIGACT News*, pages 28–48, 1986.

[170] W. Rönsch and H. Strauss. Timing results of some internal sorting algorithms on vector computers. *Parallel Computing*, 4:49–61, 1987.

[171] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computation. Technical Report 1029, CSRD, Aug. 1990.

[172] S. Saini and D. H. Bailey. NAS parallel benchmark results 12-95. Technical Report NAS-95-021, NASA Ames Research Center, Dec. 1995.

[173] S. Saini and D. H. Bailey. NAS parallel benchmark (version 1.0) results 11-96. Technical Report NAS-96-18, NASA Ames Research Center, Nov. 1996.

[174] E. J. Schwabe, G. E. Blelloch, A. Feldmann, O. Ghattas, J. Gilbert, G. Miller, D. R. O'Hallaron, J. R. Shewchuk, and S.-H. Teng. A separator-based framework for automated partitioning and mapping of parallel algorithms for numerical solution of PDEs. In *Proceedings of the 1992 DAGS/PC Symposium*, pages 48–62, June 1992.

[175] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Oct. 1996.

[176] S. D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings Supercomputing '94*, pages 97–106, Nov. 1994.

[177] T. J. Sheffler. *Match and Move, an Approach to Data Parallel Computing.* PhD thesis, School of Computer Science, Carnegie Mellon University, Oct. 1992.

[178] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, 1981.

[179] Y. Shiloach and U. Vishkin. An $O(\lg n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[180] C. Siniolakis. On the complexity of BSP sorting. Technical Report PRG-TR-9-96, Oxford University, Computing Laboratory, May 1996.

[181] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. Technical Report 96-15, Oxford University Computing Laboratory, November 1996.

[182] B. J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings International Conference on Parallel Processing*, pages 6–8, Aug. 1978.

[183] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 central processor. In *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 199–204, Oct. 1987.

[184] J. E. Smith and W. R. Taylor. Accurate modeling of interconnection networks in vector supercomputers. In *Proc. International Conference on Supercomputing*, pages 264–273, Cologne, Germany, June 1991.

[185] J. E. Smith and W. R. Taylor. Characterizing memory performance in vector multiprocessors. In *Proceedings International Conference on Supercomputing*, pages 35–44, July 1992.

[186] G. S. Sohi. High-bandwidth interleaved memories for vector processors—a simulation study. *IEEE Transactions on Computers*, 42(1):34–44, Jan. 1993.

[187] H. S. Stone. Sorting on STAR. *IEEE Transactions on Software Engineering*, 4(2):138–146, 1978.

[188] T. M. Stricker. Supporting the hypercube programming model on mesh architectures (a fast sorter for iWarp tori). In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 148–157, July 1992.

[189] P. Su. Cell algorithms on vector computers. In *Proceedings of the 1992 DAGS/PC Symposium*, pages 176–187, June 1992.

[190] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavripilis, R. Ponnusamy, and K. Crowley. PARTI primitives for unstructured and block structured problems. *Computer Systems in Engineering*, 3(1–4):73–86, Dec. 1992.

[191] R. H. Swendsen and J.-S. Wang. Nonuniversal critical dynamics in Monte Carlo simulations. *Physical Review Letters*, 58(2):86–88, Jan. 1987.

[192] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[193] V. E. Taylor, A. Ranade, and D. G. Messerschmitt. Three-dimensional finite-element analyses: implications for computer architectures. In *Proceedings Supercomputing '91*, pages 786–795, Nov. 1991.

[194] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Proceedings Supercomputing '92*, pages 14–19, Nov. 1992.

[195] Thinking Machines Corporation. *Connection Machine Scientific Software Library (CMSSL) for Fortran Version 3.1*, 1993.

[196] S. Torii, K. Kojima, Y. Kanada, A. Sakata, and S. Yoshizumi. Accelerating non-numerical processing by an extended vector processor. In *Proceedings Fourth International Conference on Data Engineering*, pages 194–201, Feb. 1988.

[197] L. W. Tucker, C. R. Feynman, and D. M. Fritzsche. Object recognition using the Connection Machine. In *Proceedings CVPR '88: the computer society conference on computer vision and pattern recognition*, pages 871–878, June 1988.

[198] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[199] T. Uehara and T. Tsuda. Benchmarking vector indirect load/store instructions. *Supercomputer*, VIII-6:57–74, Nov. 1991.

[200] L. G. Valiant. Parallelism in comparison problems. *SIAM Journal of Computing*, 4:348–355, 1975.

[201] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[202] P. J. Varman, B. R. Iyer, D. J. Haderle, and S. M. Dunn. Parallel merging: algorithm and implementation results. *Parallel Computing*, 15:165–177, 1990.

[203] U. Vishkin. Can parallel algorithms enhance serial implementation? (extended abstract). In *Proceedings 8th International Parallel Processing Symposium*, pages 376–385, 1994.

[204] H. H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7(2):170–183, June 1981.

[205] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. SPERT-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, Mar. 1996.

[206] C. W. Yang. A sparse solver with multifrontal method for general sparse linear systems on the Cray C90. In *Proceedings Fifth SIAM Conference On Applied Linear Algebra*, pages 423–427, June 1994.

[207] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, Nov. 1991.