Evaluation of Multithreading and Caching
in Large Shared Memory Parallel Computers

by

Robert Francis Boothe

B.S. (University of California at San Diego) 1985
M.S. (University of California at Berkeley) 1989

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science
in the
GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Abhiram G. Ranade, Chair
Professor David E. Culler
Professor Ronald W. Wolff

Report No. UCB/CSD-93-766
July 1993
Computer Science Division
University of California
Berkeley, CA 94720

Acknowledgements

There are many people I would like to thank for helping make my stay at Berkeley productive, rewarding, and enjoyable. First is my advisor, Abhiram Ranade, for suggesting this research area and for guiding, encouraging, and supporting my research. Next are my office mates M. T. Raghunath and Jeff Rothman, for the many discussions on research ideas, related work, politics, vegetarianism, and the densest packaging of aluminum cans. I also wish to thank my second and third readers, David Culler and Ronald Wolff, for reading through this lengthy thesis. David Culler's extensive comments helped to significantly improve the presentation of this work.

Besides those who helped influence the content and presentation of my thesis, I would also like to thank the many friends I have developed while at Berkeley. Thanks to Mike Meighan, Seth Teller, and Eric Enderton for friendship, encouragement, and appreciation of T-shirts. And thanks to the members of the Hillegass House (Ramon Caceres, Will Evans, John Hartman, Mike Hohmeyer, Steve Lucco and Ken Shirriff) for many dinners, parties, and Simpson's episodes.

Finally, I wish to thank my parents and particularly my best friend (and wife) Alison Kisch for their love, understanding, and patience during those times when we did not get to spend as much time together as we would have liked because I was busy working on this research.

This work was supported in part by the Air Force Office of Scientific Research (AFOSR/JSEP) under contract F49620-90-C-0029 and NSF Grant Number 1-442427-21936. Computing resources were provided by NSF Infrastructure Grant number CDA-8722788. Their support is gratefully acknowledged.

Abstract

Evaluation of Multithreading and Caching in Large Shared Memory Parallel Computers

by

Robert Francis Boothe
Doctor of Philosophy in Computer Science
University of California at Berkeley
Professor Abhiram G. Ranade, Chair

Shared memory multiprocessors are considered among the easiest parallel computers to program. However, building shared memory machines with thousands of processors has proven difficult. Two main problems are the long latencies to shared memory and the large network bandwidth required to support the shared memory programming style.

In this dissertation, we quantify the magnitude of these problems and evaluate multithreading and caching as mechanisms for solving them. Multithreading works by overlapping communication with computation, and caching works by filtering out a large fraction of the remote accesses.

We evaluate several multithreading models using simulations of eight benchmark applications. On systems with multithreading but without caching, we have found that the best results are obtained for the explicit-switch multithreading model. This model provides an explicit context switch instruction that allows the compiler to select the points at which context switches occur. Our results suggest that a 200 cycle memory access latency can be tolerated using multithreading levels of 10 threads or less per processor. On systems with both multithreading and caching, we have found that the switch-on-miss multithreading is best. For this model, our results suggest that a 200 cycle memory access latency can be tolerated using multithreading levels of 3 threads or less per processor.

We show that by using multithreading techniques, systems both with and without

caching are able to tolerate long latencies and achieve execution efficiencies of 80%. The difference between systems with and without caching is that the systems with caching (if properly configured) use an order of magnitude less network bandwidth. For large machines, the network cost will be a large factor in the cost of the machine, and thus the cost and complexity of cache coherency will be offset by the reduction in the network bandwidth requirement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Shared memory multiprocessors are considered among the easiest parallel computers to program [Boo89, HLRW92, Ken92, LM92, LLG^{+}92, NL91, RT86, SHG92, TKB92]. Programming is easier because the shared memory programming model allows the programmer to ignore issues such as the explicit location of data and its movement between processors. This model, however, is just an abstraction, and its success depends on the ability of the computer hardware and software to efficiently support it. This is analogous to the abstraction of a large virtual memory.

For small machines, with from 4 to 30 processors, this shared memory abstraction has been relatively easy to provide. It involves snooping caches on a single memory bus connecting all of the processors. This configuration has been named a *multi*[Bel85] and has been widely adopted for building small multiprocessors for which a single bus is able to provide sufficient bandwidth. Examples include the Sequent Symmetry[LT88], Encore Multimax[Enc87], and Silicon Graphics 4D-MP[BJS88].

However, building large shared memory machines has proven to be much more difficult than building other types of large parallel machines. For example, there exist 1,024 processor (message passing) Ncube's, 1,024 processor (message passing) CM-5's, 16,384 processor (SIMD) MasPar's, and 65,536 processor (SIMD) CM-2's[DM93]. Large commercial shared memory multiprocessors such as the KSR1[Ken92] or the Cray-T3D[KS93] have only recently been introduced and have not yet been built in very large configurations. The goal of this dissertation is to understand and address the key difficulties impeding the design and development of large shared memory multiprocessors.

1.1 The Latency Problem

By large shared memory machines, we mean hundreds or thousands of processors. For machines of this size, the bandwidth of a single bus is inadequate, and thus more complex networks, such as butterfly or grid networks, are required. The latency problem arises when a processor accesses a shared memory variable that is located in a memory module across the network. To perform this remote memory access, the processor issues a request message into the network. This message then traverses the network to the memory module. The memory module reads the value. And then it sends a result message back to the requesting processor. The interval between the sending of the request message until the return of the result message is called the *remote memory access latency*, or just *the latency*.

The networks of large machines are multi-hop networks, and messages are subject to switching, transmission, and congestion delays at each stage of the network. In a butterfly network, for example, a message traverses $O(\log p)$ nodes to reach its destination, and in a two dimensional grid network a message traverses $O(\sqrt{p})$ nodes. The aggregate latency through these networks can be hundreds of cycles. The latency becomes a problem if the processor spends a large fraction of its time sitting idle waiting for remote accesses to complete.

Figure 1.1 shows the extrapolated round trip network latencies (expressed in terms of the processor's cycle time) for several recent or proposed large parallel machines. These machines have a variety of architectures. The CM-5[LAD+92] is a message massing machine with a fat-tree[Lei85] network. DASH[LLJ+92] is a cache-coherent shared-memory multiprocessor with a two-dimensional toroidal mesh network. The KSR1[Ken92] is also a cache-coherent shared-memory multiprocessor but with a ring (or hierarchy of rings) network. And TERA[ACC+90] (expected in 1994) is a shared-memory multiprocessor without caching, and it uses a three-dimensional toroidal mesh network.

The latencies shown in the graph have been extrapolated based on scaling these networks. For the CM-5 and DASH, the latencies do not include congestion affects, and thus the actual latencies in heavily loaded networks will be higher than these curves. The KSR1 has not yet been disclosed well enough to allow extrapolating a complete latency curve.

For machines supporting 1024 processors, these graphs suggest that we can expect latencies of 200 cycles or more, once congestion affects are taken into account. Furthermore,

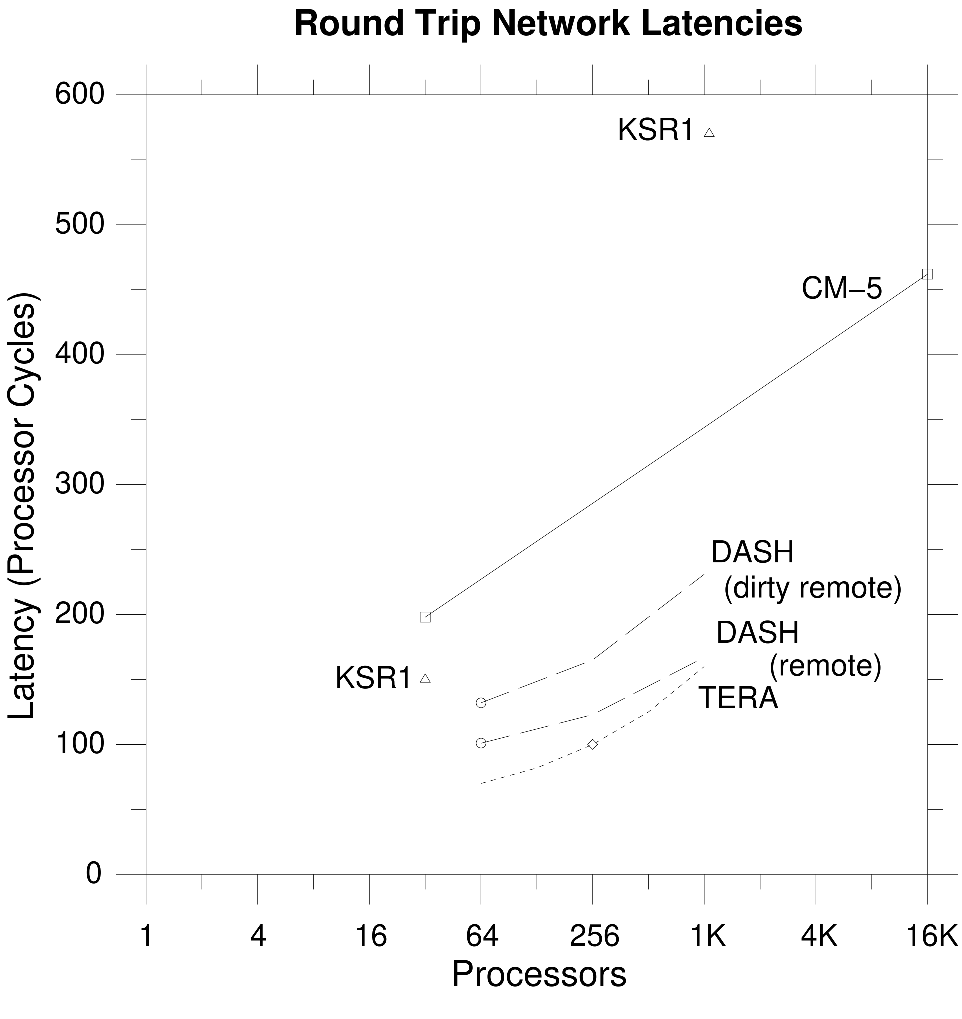


Figure 1.1: Round trip network latency as a function of machine size.

Application	cycles / remote access
sieve	11.0
blkmat	102.5
sor	4.4
ugray	17.5
water	40.2
locus	6.5
mp3d	5.7
barnes	41.3

Table 1.1: Frequency of accesses to shared memory.

we expect that the latencies will continue to increase over time as on-chip speeds are reduced more quickly than off-chip speeds.

Another way to look at these latencies is the opportunity cost in terms of the instructions that might be processed while waiting for a remote reference to return. The processors used in some of these machines execute more than one instruction per cycle[1], and thus the opportunity cost is the product of the latency by the number of instructions executable per cycle. We expect that future processors will continue to further exploit superscalar and superpipelining techniques, and thus further increase the opportunity cost of remote accesses.

In contrast with these latencies, Table 1.1 shows the reference rates for a number of parallel shared-memory applications. These applications will be used throughout this thesis and will be described in Chapter 2. These remote reference rates were calculated by dividing the total number of execution cycles (excluding stalls) by the total number of shared memory loads and stores. If these applications were run on a machine that had to stall on each remote access for a 200 cycle latency, the machine would be busy as little as 3% of the time. Clearly some means of tolerating the latency is essential for building large shared-memory parallel computers.

1.2 Solutions to the Latency Problem

Figure 1.2 lists six mechanisms for addressing the latency problem. Caching[Smi82] works by filtering out redundant references. Multithreading[Kow85] allows switching to a

[1] The KSR1 processor is two-way superscalar, and the TERA processor has a wide instruction word and can issue three operations per cycle.

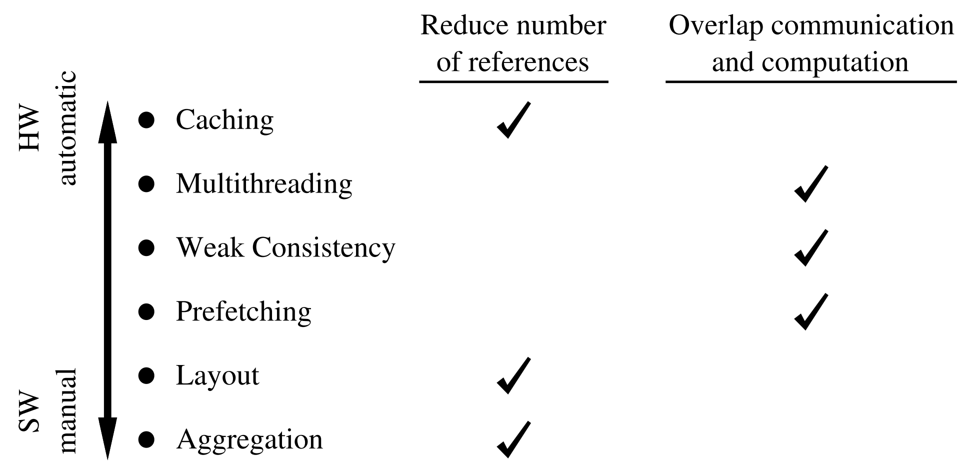


Figure 1.2: Mechanisms for reducing the impact of memory latency.

different thread while waiting for a remote reference to complete. Weak consistency[AH90] allows some overlapping of writes without concern that they might arrive out of order. Prefetching[LYL87] allows requesting data before it will be needed. By layout[Hig93] we mean the idea of arranging data on or near the processor that is going to use it. And aggregation[HLRW92] is the idea of getting large amounts of data at once.

The mechanisms near the top of the diagram are more commonly automatic (or invisible) as far as the programmer is concerned and are generally implemented in hardware. The mechanisms near the bottom of the diagram are often implemented in software either by a smart compiler or manually by the programmer. For example in a message passing program, the programmer explicitly specifies the layout of data and the packaging of messages(i.e., aggregation of data).

All of these mechanisms have their limitations. Caches must be kept coherent, which becomes complex for large machines[HLRW92, TD91]. Furthermore, the hit rates may be low for accesses to shared data[DRPS87, GHG$^+$91]. Multithreading requires complex hardware to allow rapidly switching between the threads on a processor. And since it requires extra threads, it also requires extra parallelism and is thus limited to larger problems. Consistency models prohibit many compiler optimizations. Weak consistency allows more than sequential consistency, but it is a less intuitive programming model[GLL$^+$90]. Prefetching is useful for applications with predictable behavior such as many scientific codes, but it is of limited applicability for more chaotic codes that use complex data structures. It

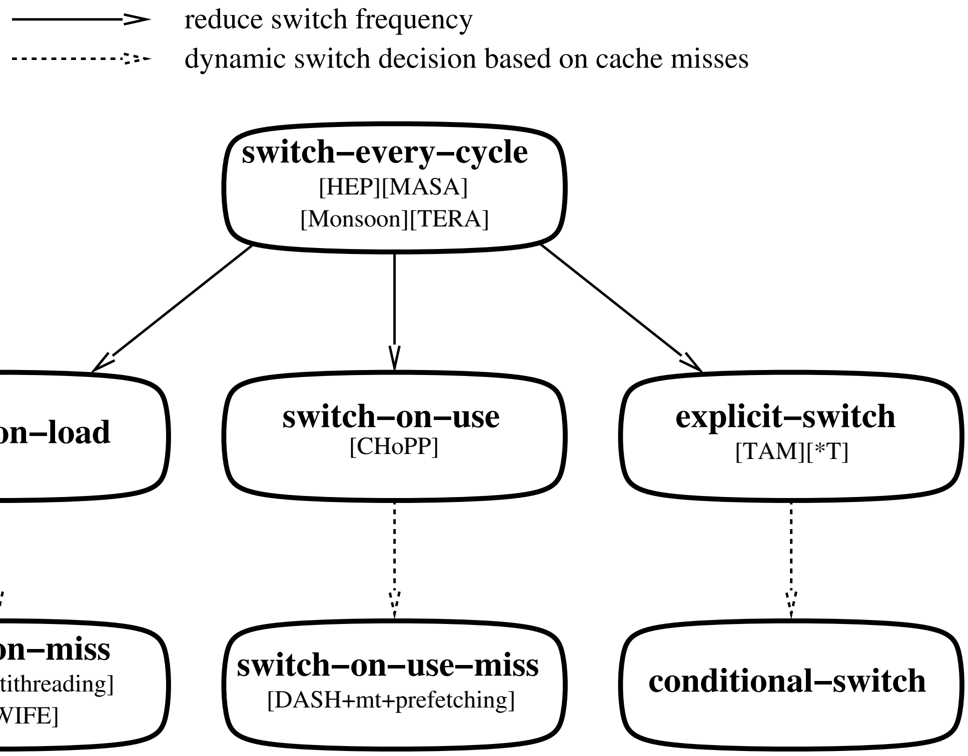


Figure 1.3: Evolution of multithreading models

also may waste bandwidth by prefetching data that is not used. Generation of good layouts is limited to the more regular and predictable applications. Finally, aggregation requires coarse grain parallelism where large data items can be manipulated.

In this dissertation we have chosen to focus on evaluating the mechanisms of multithreading and caching. A weakly consistent memory model is assumed throughout. These are the more automatic mechanisms and are most consistent with the shared memory programming model. The mechanisms of prefetching and layout can be of additional benefit, and we have incorporated them in a limited fashion in a few of our studies, however there remains room for further research in these areas.

1.3 Overview of Previous Multithreading Work

Previous multithreading research has been motivated by three concerns: tolerating memory latency, building a fast pipeline, and supporting a dynamic dataflow like execution model. Figure 1.3 shows the evolution of multithreading models and some of the motivations for moving from one model to another. Some of these models have not been studied previously but can be predicted based on the motivations.

The costs and concerns of multithreading are: the large number of threads needed, the scheduling mechanism used to schedule the many threads on a processor, the cycles lost to context switching overhead, and the large register file. These costs and concerns are influenced by when and how often context switching is performed.

1.3.1 Fast Pipeline

The oldest model **switch-every-cycle** was used in the Denelcor HEP [Kow85] and in MASA [HF88]. After each instruction, the processor switches to a different thread. This allows a fast CPU pipeline to be built because it eliminates data dependencies between instructions in the pipeline by interleaving different threads. It also allows memory latencies to be tolerated by not scheduling a thread until its reference has completed. Unfortunately this model requires a large number of threads and a large amount of hardware to support them. Also, by interleaving the instructions from many threads, a single thread is limited to a small fraction of the processing power. TERA[ACC+90] is similar to the HEP, but the scheduling policy has been changed so that a thread can issue more than one reference into the network before waiting for the results.

1.3.2 Hiding Memory Latency

The rest of the multithreading models that we consider execute a thread for many cycles before context switching. The optimizing compiler is responsible for the ordering of instructions so as to hide the small pipeline delays, and context switches are thus used only to hide the long memory latency of remote accesses.

The **switch-on-load** model switches on load instructions which access shared memory. Loads from local memory and other instructions all complete quickly and can be scheduled by the compiler. Shared memory stores do not wait for their completion and therefore do not cause context switches either. The advantages of this model over **switch-every-cycle** are that a single thread can execute at full speed until it context switches, and fewer total threads will be needed since multithreading is not being used to hide pipeline delays. Simpler hardware may also be possible since context switches are less frequent.

The **switch-on-load** model sometimes context switches sooner than it needs to.

If a compiler can order instructions so that a load is issued several cycles before the value is used, the context switch does not have to occur until the actual use of the value. This allows a thread to hide some of its own memory latency by prefetching data. This can be implemented by adding a *valid* bit to each register. The valid bit is cleared when the load instruction is issued and set when the result returns from the network. The **switch-on-use** multithreading model context switches on the use of an invalid register, rather than on a `load` instruction. The **switch-on-use** model was used in the design of the CHoPP architecture[MPS87] which had a VLIW multithreaded processor.

A benefit of the **switch-on-use** model is that several load instructions can be grouped together so that all of the loads in the group can be issued into the memory network before any of the results are used. This allows a single context switch (on the first of the uses) to wait for all of the loads in the group. For example, a simple computation may load two values from shared memory and then compute their average. Both loads should be issued into the memory network and then a single context switch performed upon the use of the first value. This allows waiting for both loads together rather than individually.

An alternative method for grouping shared loads together is to add an explicit context switch instruction between the group of loads and their subsequent uses. The **explicit-switch** model allows similar grouping to **switch-on-use**, but is simpler to implement and requires the addition of only a single instruction. We evaluate the **explicit-switch** model in Chapter 4 and find that it can eliminate from 50% to 80% of the context switches needed by the **switch-on-load** model.

The most recent data flow research[CSS+91, NPA92] has adopted the **explicit-switch** model. Short threads execute until their completion at which point they cause a context switch to a new thread.

1.3.3 Adding Caches

By adding caches to the previous models, the cache can satisfy many of the shared loads without going to shared memory. Only those shared loads that miss in the cache will have long latencies and cause context switches. By filtering out many of the remote references, caching makes the job of hiding the memory latency easier, and fewer threads will be needed to cover the latency. In Chapter 5 our results will show that for most applications just 2 or 3 threads per processor is sufficient.

The **switch-on-miss** model switches at points where load instructions miss in the cache. An early study of this by Weber & Gupta[WG89] suggested substantial performance benefits were available, but a later study as part of the DASH project [GHG+91] had less optimistic results. **Switch-on-miss** multithreading was also studied as part of the ALEWIFE project [ALKK90] and achieved good results for a few simple applications. One draw back of context-switching on cache misses is that the context switch is detected after a number of subsequent instructions have started down the CPU pipeline. These instructions must be canceled, and thus there will be a context switch cost of several cycles because of the wasted pipeline slots.

The **switch-on-use-miss** model context switches when a `use` instruction tries to use the value from a shared load that missed in the cache. It was studied (approximately) by the DASH project[GHG+91] when they looked at the combination of prefetching and multithreading. Their prefetch instructions act like the initial load instructions, and their subsequent load instructions act like the use of the data. They found little benefit from prefetching when combined with multithreading, however they state that their prefetching method was meant for a single threaded processor and should be done differently for a multithreaded processor.

The **conditional-switch** model adds caching to the **explicit-switch** model. The code appears the same as that for the explicit-switch model: there is a group of load instructions, followed by a context switch instruction, followed by the instructions that use the loaded data. The difference is that the context switch instruction is treated as a *conditional* switch instruction. If any of the loads preceding the switch instruction missed in the cache, a context switch is performed as expected. But if all of the preceding loads hit, the context switch instruction is ignored and the thread continues executing. This model provides the benefits of grouping and caching as in the **switch-on-use-miss** model, but it may be simpler to implement.

1.4 Limited Bandwidth

Besides having long latencies on remote accesses, the networks on large parallel machines are also likely to have limited bisection bandwidths[2]. Figure 1.4 shows the bi-

[2] Bisection bandwidth is defined as the minimum bandwidth capacity between the two halves of a bisected machines, considering all possible bisections.

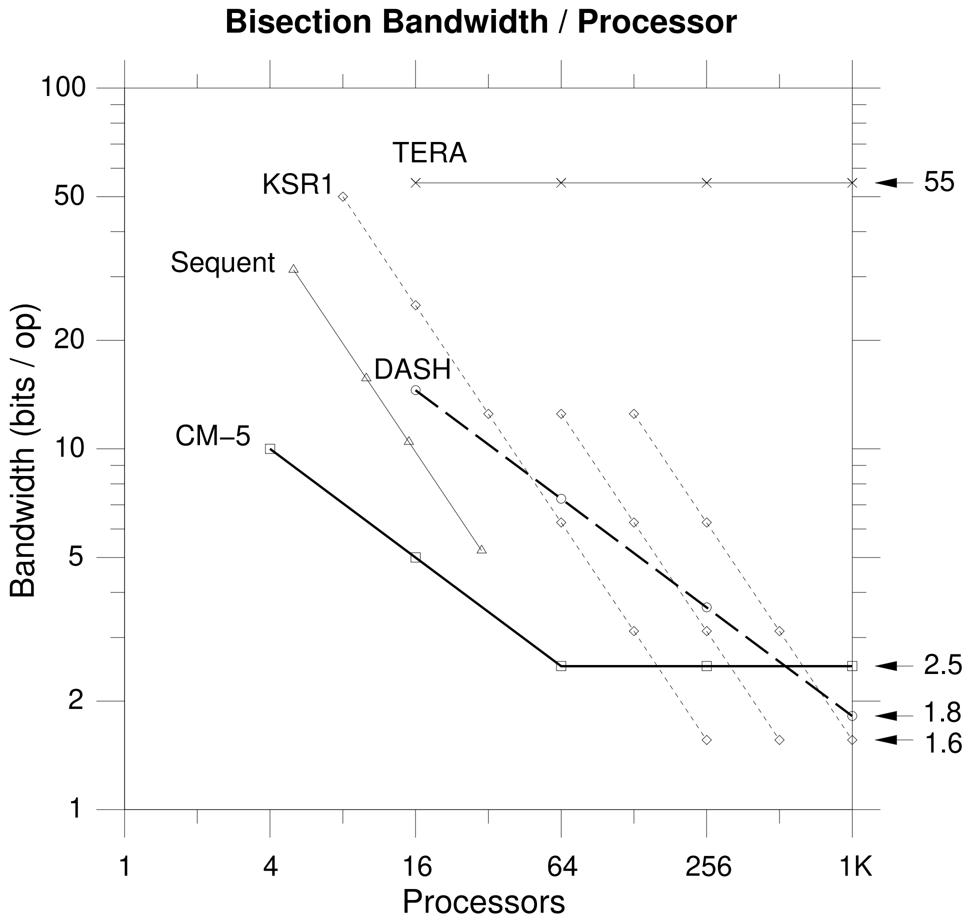


Figure 1.4: Bisection bandwidth of various parallel computers as a function of machine size. The bandwidth is expressed in terms of bits per processor operation at the peak capacity and peak execution rate.

section bandwidths of various parallel computers. These are peak bandwidths and are not expected to be fully achieved under real traffic patterns. Also, the bandwidths have been extrapolated to 1024 processors based on the proposed designs, even if such a large machine has not actually been built. The bandwidth values were calculated based on descriptions of the networks in [ML92], [Ken92], [LLJ+92], [LAD+92], and [ACC+90].

The key point of this graph is that for most networks, the bisection bandwidth drops as the number processors is increased, and that for a large (1024 processor) machine, only 1 or 2 bits of bandwidth per operation will be available. In fact, achievable bandwidth may be only half of that amount because of the congestion caused by irregular traffic patterns[Dal90].

The reason the bandwidth drops off as the number of processors increases is related to the scaling characteristics of the networks. For the Sequent[ML92], there is a single shared bus and thus the bandwidth per processor diminishes in proportion to the number of processors. For both bandwidth and electrical reasons, sharing a single bus limits the number of processors to around 30.

The KSR1[Ken92] is similar, it uses a single high bandwidth ring for small machines, or a two level ring of rings for larger machines. The bisection bandwidth depends only on the top level ring, and when calculated on a per processor basis, decreases linearly. For large machines they stave off the bandwidth decline by providing multiple rings at the top level. The three lines in Figure 1.4 for the KSR1 represent the three configuration options for these top level rings. The largest option has 4 GB/sec of bandwidth along the ring (8 GB/sec crossing the bisection), but when divided among 1024 processors (two-way superscalar) running at 20 Mhz, this provides only 1.6 bits per operation.

The DASH[LLJ+92] architecture scales better because it is based on a 2-D wrap-around mesh (torus) rather than a ring. The bisection bandwidth per processor drop off as the square root of the number of processors. At 1024 processors, which is more than this design was meant for, the bisection bandwidth is 1.8 bits per operation.

The CM-5 network[LAD+92] is a fat tree. Fat trees[Lei85] are a family of networks where the connections between nodes at higher levels of the tree are generally "fatter" than the connections between nodes at lower levels of the tree. With the appropriate connection widths, a fat tree can provide constant bisection bandwidth per processor as the machine is scaled. However to save costs, the designers chose to eliminate many of the channels at the higher levels in the tree. For 1024 processors, the bisection bandwidth is 2.5 bits per operation. This figure is without the optional vector units. If they are included and achieve their potential factor of 4 performance increase, the bandwidth when expressed in bits per operation will be reduced to only 0.6 bits.

Finally, and in contrast to the other networks, the proposed Tera network provides a bisection bandwidth of 55 bits per operation and scales the bandwidth linearly with the number of processors. The network is a sparsely populated 3-D wrap-around mesh[ACC+90], and to scale the bandwidth linearly, they increase the number of network nodes faster than the number of processors. For large machines Tera has more than an order of magnitude greater bandwidth than other machines. We suspect however, that providing this large network bandwidth may not prove cost effective.

In this thesis we do not focus on any particular network topology. Instead we measure the bandwidth needs of our benchmark applications, and then use the results to reason about the types of machines that should be built and the bandwidth capacity that they should supply.

1.5 Overview of Thesis

In this dissertation we concentrate on the **switch-on-load**, **explicit-switch**, **switch-on-miss**, and **conditional-switch** models. If caches are not used, our results will show that grouping is important and thus **explicit-switch** is preferable to **switch-on-load**. However if caches *are* used, our results will show that grouping has little benefit and thus **switch-on-miss** is preferable to **conditional-switch**.

The remainder of this dissertation is organized as follows: Chapter 2 discusses our simulation methodology and our set of benchmark applications. Chapter 3 presents a performance model for a multithreaded processor. Chapter 4 focuses on hiding latency with multithreading and evaluates the **switch-on-load** and **explicit-switch** multithreading models. Chapter 5 adds coherent caching to the system and evaluates the **switch-on-miss** and **conditional-switch** multithreading models. Chapter 6 considers the problem of limited network bandwidth and presents results on the amount of bandwidth that is needed by the various applications and multithreading models. Chapter 7 presents miscellaneous studies and experiments on synchronization and various caching issues. Chapter 8 discusses the hardware mechanisms needed to support a multithreaded processor. And Chapter 9 presents conclusions and directions for future research.

There are also two appendices: Appendix A explains a new method for plotting distributions that we have introduced in order to visually present both clearly and compactly the types of distributions that we have encountered. And Appendix B explains the techniques used to build the simulator that made this research possible.

Chapter 2

Methodology

This chapter discusses our simulation based research methodology. First we present our model of a large shared memory parallel machine. Then we present the benchmark applications that we have used to evaluate this model. And lastly we present our simulation system and its limitations.

We have chosen simulation over analytic modeling because parallel programs are complex interacting systems that are difficult to model accurately. Saavedra-Barrera *et al.*[SBCvE90, SBC91] and Agarwal[Aga92] have done analytic models of simplified multithreading systems, but for reasons of tractability they are forced to assume that threads are independent and have accesses at exponentially distributed intervals (i.e. according to a Poisson process). The threads of parallel programs, however, are not independent because they have substantial sharing of data, and they use synchronization to coordinate this sharing. We will also see in Section 3.2 that many programs have distinctly non-exponential inter-arrival distributions. Real programs also have many parameters (such as reference rates, amount of sharing, synchronization patterns, imperfect load balancing, parallelization overhead, and time varying behavior) that are difficult to characterize and are not yet well understood.

2.1 Machine Model

Figure 2.1 shows our model of a large shared memory multiprocessor. It consists of a number of processors and a like number of shared memory modules that are all interconnected by a switching network. Each processor also has a local memory which holds

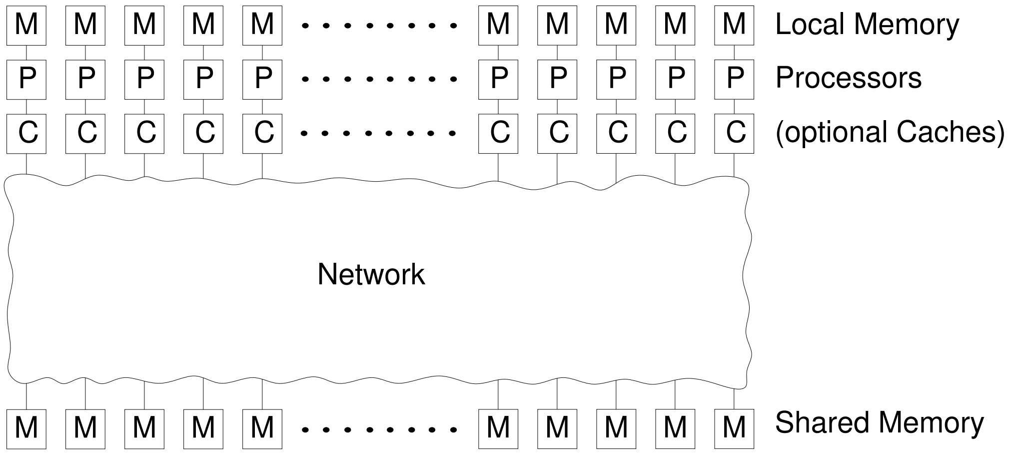


Figure 2.1: Model of large shared memory multiprocessor.

variables local to threads, stacks, and the code. We assume all accesses to local memory are instantaneous. This is reasonable since local data and instructions can be easily cached, and any misses can be serviced locally. Accesses to shared memory are sent across the network and thus have a long latency before they return.

We look at two variations of this model: one with caching of shared data (as shown in the figure), and the other without it. Both types of systems have been and are being built. Examples of systems without coherent caching are the HEP[Kow85], BBN Butterfly[BBN89], and Tera[ACC+90]. Systems with coherent caching include all of the shared bus based systems such as the Sequent[Ost89] as well as the more scalable KSR1[Ken92] and research projects such as DASH[LLG+90, LLG+92, LLJ+92] and ALEWIFE[ALKK90].

For a real machine, the model in Figure 2.1 will likely be folded back upon itself so that each processor is directly connected to one of the shared memory modules. This would give each processor direct access to a small portion of shared memory. If the programmer (or compiler) can control the layout of data onto the memory modules, she might be able to arrange the data on or near the processors where it will be used. Such layouts could eliminate a large fraction of the remote references, but they are not possible for many applications[SHG92].

In this research (with our applications, programming languages, and compiler technology) we do not have the capability of customizing the data layout for each application. We therefore assume that data is randomly interleaved across the memory modules. This interleaving is done at the level of the largest memory access unit. For systems without caches, the largest memory access unit is a double word; for systems with caches, the memory access unit is a cache line.

Without optimization of the data layout, the performance advantage of the folded machine configuration is small. On a 1000 processor machine, for example, only 1/1000th of the accesses will be to the locally accessible memory module. This small factor is insignificant, and thus we study the model as shown in Figure 2.1 (with no locally accessible shared memory.)

Mellor-Crummey and Scott[MCS91] argue against building such "dance hall"[1] machines because their efficient synchronization techniques depend upon having either coherent caching or local access to part of shared memory. In Section 7.1 we propose preferable synchronization techniques that will eliminate this taboo on "dance hall" machines.

2.1.1 Network

There are many proposed network topologies[2]. Figure 2.2 shows some that are popular for reasons involving: latency, bandwidth, cost, modularity, and availability of simple routing algorithms. Network design is still an active research area with many competing concerns. In this research we do not select any specific network, but instead we focus on the general characteristics of all of these networks. First, they all are packet switched networks that allow many parallel references to be traveling through the network simultaneously. Second, if the processors can support it, each processor may have several outstanding references in the network at once. And third, references will have long latencies because they are routed through many network nodes and experience congestion and delays along the way.

In this research we are interested in machines which range in size from a hundred processors up to a few thousand processors. We will use the design point of a 1000 processor machine for choosing parameters and further reasoning. Figure 1.1 in Chapter 1 showed the latencies of several existing and proposed large interconnection networks. For a 1000 processor machine, these networks all have round trip latencies in the range of a few hundreds processor cycles. We choose a latency of 200 cycles as representative of these figures

[1]The term "dance hall" comes from an analogy between a machine with the processors and memories separated at different ends of the network and a dance hall with separated boys and girls.

[2]See for example: Almasi/Gottlieb[AG89] chapter 8.

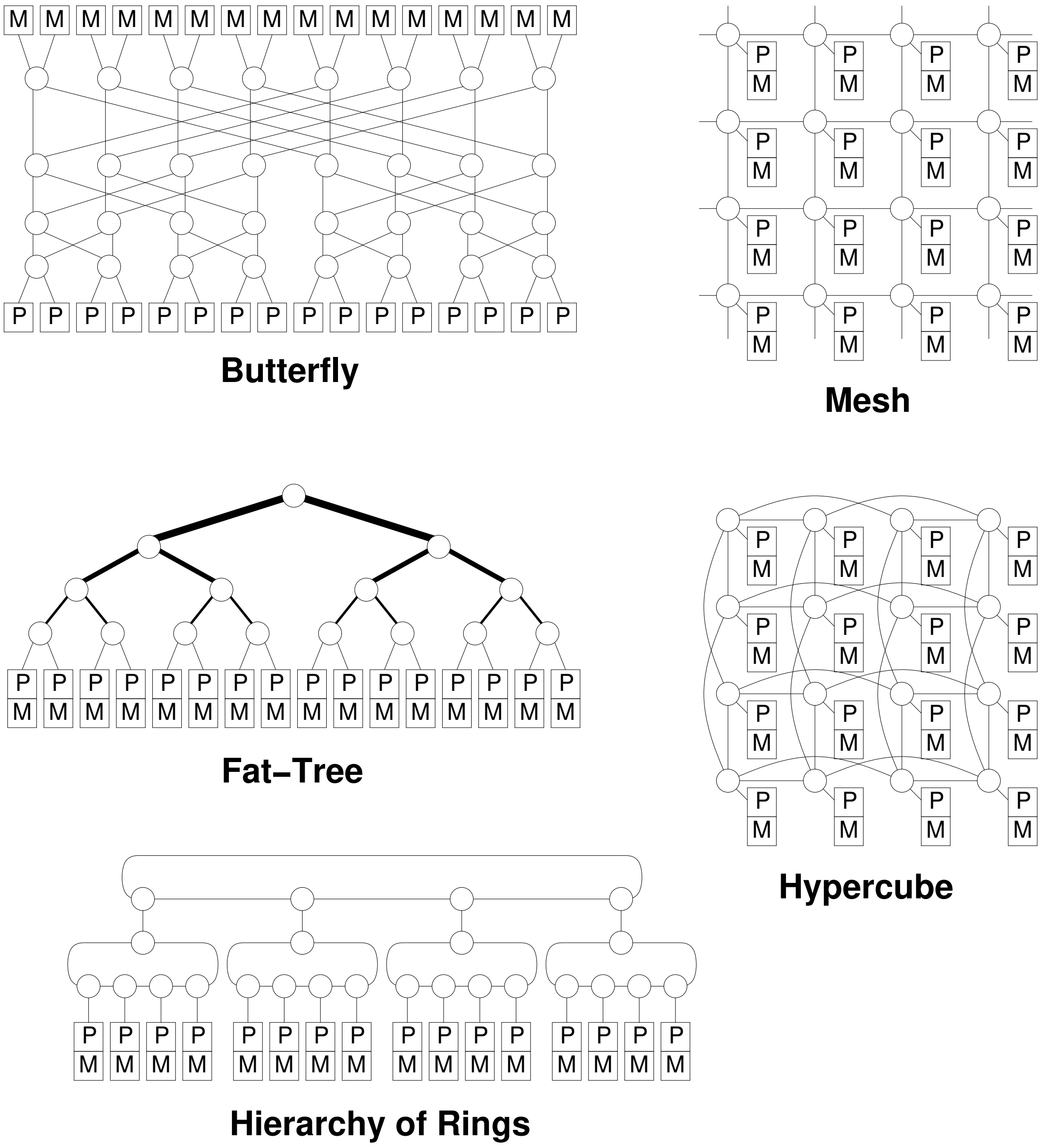


Figure 2.2: Some popular network topologies.

and focus this research on tolerating latencies of this magnitude.

We model the network and shared memories simply as a black box that takes 200 cycles to respond to a remote memory reference. In a real network, much of this latency will come from delays due to minor random congestion, and these small delays are an expected component of the 200 cycle latency. A more difficult case is severe congestion caused, for instance, by a program induced hot spot in which every processor tries to simultaneously access the same memory location. In this case delays can become much longer than the delay of 200 cycles that we have assumed. We ignore such congestion initially, but in Section 6.2 we assess the frequency of such hot spots and their impact on our simulation results.

2.1.2 Processor

We expect that the processors used in parallel systems will be the same or very similar to the microprocessors used in high performance workstations. This is because the peak performance of a parallel system is the product of the performance of a single processor and the number of processors. Such a large development effort is put into the race for the highest performance microprocessor that these push the technology curve and offer the most cost effective single processor.

To tolerate latency, however, we evaluate multithreading techniques which require that the processor be able to context switch rapidly between threads. In the past, multithreaded processors, such as the HEP[Kow85], have involved very different and complex processor designs that context switch every cycle and use a a large number of threads. Instead we look at multithreaded processors that are similar to today's RISC microprocessors with the addition of being able to context switch on long latency remote memory accesses. Chapter 8 discusses the hardware issues in detail. Here we wish simply to specify our assumptions about the multithreaded processors that we simulate, and leave their justification to Chapter 8.

We assume the same instruction set and instruction timings as the MIPS R3000[Kan89], but with a few modifications. Most importantly, we assume that the register file has been replicated on the chip enough times so that each thread running on the chip can have its own set of registers. Because the registers are on chip and do not have to be saved or loaded from memory on a context switch, the processor should be able to switch quickly between threads; in some cases as fast a single cycle (see Section 8.1.1).

Another modification is that we have added double word loads and stores to the instruction set. Many floating point numbers are stored as double words, and it is crucial (when the network has long latencies) to get the whole thing at once rather than having two separate references as is done on the MIPS R3000. More recent machines, such as the MIPS R4000, all provide double word loads and stores.

Finally, we provide both local and shared versions of all memory access instructions. This is based on the assumption that memory references can be classified by the compiler as either local or shared. For instance references to locations in a shared array would use `shared-load` instructions while references to local variables would use `local-load` instructions. This compiler classification may not be possible in the case of pointers if it is unclear what is pointed to and whether or not it resides in shared or local memory. We call these unclear cases *ambiguous* pointers, and they must be resolved at run-time either with extra code or special hardware, which will likely slow down and/or complicate the machine. Ideally we would like the compiler to classify as many references as possible because this information will be needed for compiler optimizations in Chapter 4.

2.1.3 Programming Language

Our applications are written in the augmented C dialect that is used in writing shared memory programs on the Sequent[Ost89]. Figure 2.3 shows an example of a simple program that multiplies two matrices. The arrays `d`, `e`, and `f` are declared as residing in shared memory by the addition of the type modifier "`shared`" before their declaration.

Unfortunately this language does not have shared memory declarations for objects accessed indirectly via pointers. The compiler does not know that the parameters `a`, `b`, and `c` to the `worker` function will be arrays in shared memory. For this simple program the compiler might deduce this information through global analysis, but in the general case this is difficult.

In our simulations we at first used dynamic testing of pointers to determine if addresses were in local or shared memory. Later we observed that for our application programs, true ambiguous cases (where sometimes a pointer points to a local location but at other times points to a shared location) never occurred. We thus collected classification information from an initial run of the application and fed it back into subsequent compilations, as is done in trace analysis. This allowed complete compiler classification of all

```
/*----------------------------------------------------------------
    simple example program that multiplies two matrices
----------------------------------------------------------------*/
shared double d[10][10], e[10][10], f[10][10];

main()                           /* will compute f = d * e */
{
  ... initialize: d and e ...
  m_set_procs(100);              /* set to 100 threads */
  m_fork(worker, d, e, f);       /* fork the threads */
  ... print result: f ...
}

worker(a, b, c)                  /* will compute c = a * b */
  double a[10][10], b[10][10], c[10][10];
{
  int i, j, k, myid;
  double sum;

  myid = m_get_myid();           /* get unique thread id [0,100) */
  i = myid/10;                   /* calculate thread's row */
  j = myid%10;                   /* calculate thread's col */

  sum = 0.0;
  for (k = 0; k < 10; k++)
    sum += a[i][k] * b[k][j];    /* calculate sum of products */
  c[i][j] = sum;                 /* each thread calculates */
}                                /*  one element of c      */
```

Figure 2.3: Example of a shared memory program.

Application	Lines	Cycles	Description & Problem size
sieve	236	106 M	counts primes number of primes < 4,000,000
blkmat	369	87 M	blocked matrix multiply 200×200 matrices
sor	333	258 M	successive over relaxation 192×192 grid
ugray	10784	1353 M	ray tracing graphics renderer gears (7169 faces), 20×512 slice of image
water	1368	1082 M	simulate a system of water molecules 343 molecules, 2 iterations
locus	6347	665 M	route wires in a standard cell circuit Primary2 (1290 cells $\times$ 20 channels)
mp3d	1510	192 M	simulate rarefied hypersonic flow 100,000 particles, 10 iterations
barnes	2109	1148 M	gravitational N-body simulation 4096 bodies in two clusters

Table 2.1: Parallel Applications

references.

Although we were able to obtain complete classification information through trace analysis, we would like to advocate that this shared versus local distinction is important and that it should be supported explicitly by future shared memory parallel languages. This could be done by allowing declarations of parameters (such as `a`, `b`, and `c` in the example) and pointers as *pointing to shared memory.*

2.2 Benchmark Applications

Table 2.1 shows the eight benchmark applications used in this research. These are all scientific programs that perform some computation or numeric simulation. The first three (`sieve`, `blkmat`, and `sor`) are toy applications written as part of this research. The other five are real applications. `Ugray`[Boo89] was parallelized by myself, and has been used in a few parallelism studies[BR92, LS91, O'K92]. The last four (`water`, `mp3d`, `locus`, and `barnes`) are part of the Stanford SPLASH benchmark set[SWG92] and have been used in many studies, especially those associated with the DASH project[LLJ$^+$92].

Each of the applications has some unique behavioral characteristic(s), and the three toy applications were chosen because they each have distinct behaviors that broaden

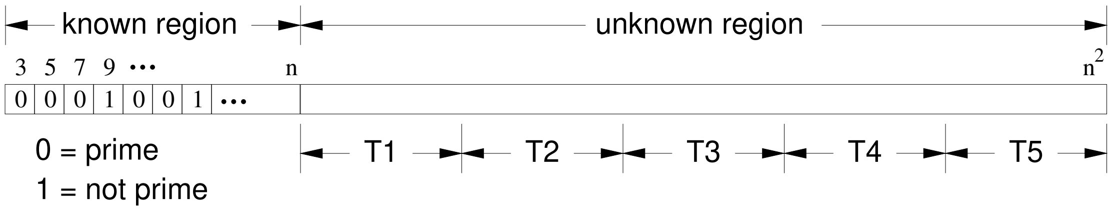


Figure 2.4: Sieve: a parallel primes finder.

the scope of the total benchmark set. The following sections contain brief descriptions of each application and report their main characteristics. Their original descriptions and codes often mix the terminology of *processes* and *processors*. Here we are more careful and will use the term *thread* to mean a *process*, and *processor* to mean a physical processor.

2.2.1 Sieve

The `sieve` application finds and counts the number of primes that are less than some given number. Figure 2.4 shows how this algorithm was partitioned for parallel execution. It represents the number space by a bit vector in shared memory with one bit for each odd number (the number 2 is treated separately). Initially all the bits are 0, which means that the numbers might be prime. As the sieve executes, whenever a number is determined to not be prime, its bit is set to 1.

Initially there is a small region of known primes. If this region goes up to n, then it is adequate for computing all primes up to n^2. The region from n to n^2 is called the unknown region, and it is partitioned across the threads. Each thread uses the primes in the known region to perform the sieve in its portion of the unknown region. A barrier synchronization is then done, and the known region is expand to n^2 and the unknown region become n^2 to n^4.

The main characteristics of `sieve` are:

- Regular intervals between shared memory accesses.
- No sharing of data in the unknown region.
- Very infrequent synchronization.

2.2.2 Blkmat

The `blkmat` application multiplies matrices. Figure 2.5(a) shows the partitioning

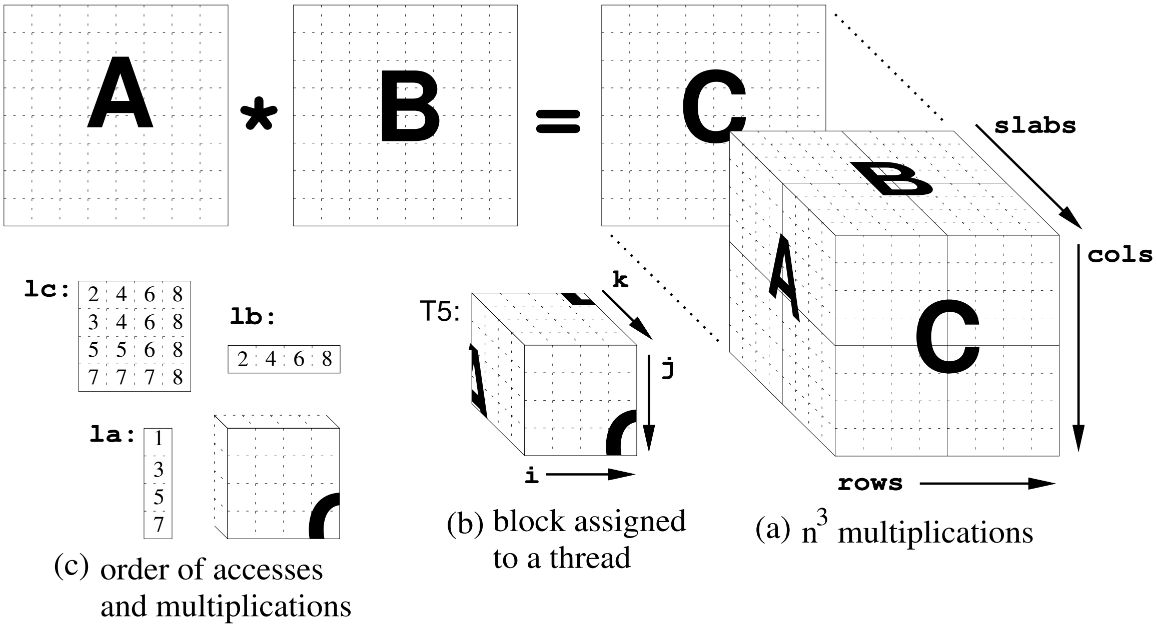
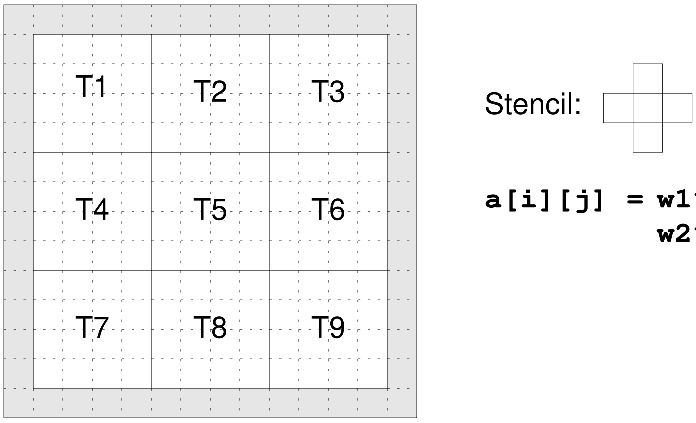


Figure 2.5: Blkmat: blocked matrix multiply.

of the computation for parallel execution. Multiplying two $n \times n$ matrices uses a total of n^3 multiplications in the standard algorithm, which we have used. This set of multiplications is then partitioned along all three dimensions so as to minimize the surface to volume ratio of the partitions. Surface area represents the amount of inherent communication that must be done, because it is the portion of the arrays A, B, and C that must be accessed by a thread. The volume represents the amount of computation that is performed. Partitioning along all three dimensions in cuboidal blocks minimizes the ratio of communication to computation.

Figure 2.5(b) shows the 4×4 block of the computation assigned to thread 5. To do its computations, thread 5 will read values from a 4×4 region of matrix A and a 4×4 region of matrix B. To achieve the minimum communication level, the program copies the values from A and B into local arrays `la` and `lb` and reuses these local copies rather than repeatedly retrieving the same values from shared memory. The order of reading values into `la` and `lb` and the order of computation of `lc` is shown by the numbers in Figure 2.5(c), for one slice of the block. This unusual ordering was used so as to interleave the computation and communication as much as possible.

At the end of the computation, locking is used to coordinate the accumulation of partial sums into the matrix C. Since this involves little computation, the access rates




```
a[i][j] = w1*a[i][j] +
          w2*(a[i+1][j] +
              a[i-1][j] +
              a[i][j+1] +
              a[i][j-1]);
```

Figure 2.6: Sor: successive over relaxation.

during this phase are much higher than in the main calculation phase of the algorithm.

The main characteristics of `blkmat` are:

- Low access rates to shared memory.
- Varying intervals between accesses.
- A separate termination phase with much higher access rates.

2.2.3 Sor

The `sor` application is an iterative solver of Laplace's equation using the method of successive over relaxation. We use it for computing the heat flow in a square metal plate. The plate is represented by a grid of cells and is partitioned into regions as shown in Figure 2.6. Interactions between threads occur along the edges between regions, and thus the cells are partitioned into squarish regions in order to minimize the lengths of their edges. The outside edges of the grid contain the fixed boundary conditions and are not part of any thread's partition.

The computation proceeds by taking a cell in the grid and replacing it with a new value computed as a weighted sum of the old value and the four manhattan neighboring cells. The weights are chosen so as to make the computation converge as quickly as possible. After every few iterations convergence is checked for by comparing the new values to saved copies of previous values.

In order to avoid mixing results from the current and previous iterations, the grid is split like a checkerboard into red and black cells. First all of the red cells are calculated and updated, and then this is followed by a barrier synchronization. After the barrier, all of the black cells are calculated and updated, and then there is another barrier. Recalculation of the red and black cells alternates in this fashion.

The main characteristics of `sor` are:

- High access rates to shared memory.
- Repeated barrier synchronization.
- Static partitioning and reuse of shared data.

2.2.4 Ugray

The `ugray` application is a ray tracing graphics renderer. This is a computationally intensive rendering algorithm for producing high quality images. The sequential program is discussed in [Mar87], and its parallelization is discussed in [Boo89].

The data structures used to describe a scene are a complex web of cross connected structures. The top level is a large three dimensional array which is a coarse cellular map of space. It is used to quickly locate objects as light rays are traced through and bounce around space. Each cells has a linked list of objects that intersect it, and objects themselves are circular linked lists of vertices with additional links to attributes. The test scene of glass gears has 7169 faces and uses 7 Megabytes of data structures to describe it.

The main characteristics of `ugray` are:

- Complex linked data structures. (can not be prefetched)
- Moderate access rates. (complex calculations)
- Dynamic scheduling of jobs.
- Unpredictable reuse of data.

2.2.5 Water

The `water` application simulates a system of water molecules in the liquid state. A brief description of this application and its parallelization appears in the SPLASH report[SWG92]. It is an N-body simulator that computes the pairwise interactions between molecules, except that is uses a spherical cut-off radius and thus ignores interactions with molecules beyond a certain distance.

The data set we used has 343 molecules (this was the largest data set available). These molecules are statically subdivided among the threads, and the same molecules are kept from iteration to iteration regardless of whether or not they are close to each other (for load balancing reasons it was actually better if they were not close to each other). Because of the small (and odd) number of molecules, load balancing was usually imperfect. The most heavily loaded thread determines the rate of progress of the computation. Thus for our simulations we tried to choose the number of threads so that any imbalanced threads would have less work, rather than more work, compared to the bulk of the threads. Good values for the number of threads are: 343, 172, 115, 86, 69, 58, 49, etc. With 172 threads, for example, each thread is given 2 molecules except for the last thread which gets only 1. Thus the last thread will finish earlier than the others and have to wait. In contrast, if instead we had used 171 threads, each thread would get 2 molecules except for one of them which would get 3 molecules! This thread would be the slowpoke and all the others would have to wait on it. Nearly the entire machine would sit idle for a third of the time.

This application's writers were conscious of remote reference concerns and ordered the computations so as to quickly reuse data and thereby obtain good cache hit rates. There are also some large calculations that use only local variables and thus there are long periods during which no remote accesses occur.

The main characteristics of `water` are:

- Bursty traffic with long periods having no remote accesses.
- Good reuse of data.
- Imperfect static load balancing that is sensitive to the number of threads used.

2.2.6 Locus

The `locus` application is a router for standard cell VLSI circuits. A brief description of this application and its parallelization appears in the SPLASH report[SWG92].

The main task is to route wires through a grid of horizontal and vertical channels, where a good route is one that minimizes the number of wires passing through heavily used channels. The wire counts of the channels are maintained in a two dimensional cost array, and the bulk of the the shared memory accesses come from the evaluation of possible routes through this array. These route evaluations typically involve reading and comparing linear sequences of horizontal or vertical array elements.

Wires to be routed are put on several shared queues, and the threads dynamically schedule and route wires from these queues until the circuit is completely routed. After this initial routing, wires are ripped up and re-routed to further optimize the result.

We used the largest input circuit available (Primary2.grin) which has 3817 wires and a 1290×20 array of routing channels. This input shows good speedups up to around 64 threads, but performance gains diminish past this point. This application has the least parallelism of the applications we have used, but was included for reasons of application diversity.

The main characteristics of `locus` are:

- High access rates.
- Linear sequences of array accesses.
- Dynamic scheduling.
- Limited parallelism.

2.2.7 Mp3d

The `mp3d` application simulates rarefied fluid flow, such as that which occurs in the upper atmosphere. It uses Monte Carlo methods and simulates a representative collection of molecules. A brief description of the application and its parallelization appears in the SPLASH report[SWG92].

We simulate a system of 100,000 molecules. These molecules are statically assigned to threads, but no attempt is made to assign nearby molecules to the same thread. Because of this, the interactions of molecules are almost always with molecules assigned to other threads, and since the molecule are all moving, the collection of interactions is changing constantly. The net result is that there is little reuse of data.[3]

The main characteristics of `mp3d` are:

- High access rates.
- Little reuse of data.

2.2.8 Barnes

The `barnes` application simulates the gravitational interaction of a system of n bodies. It uses the $O(n \log n)$ Barnes-Hut algorithm rather than the $O(n^2)$ direct pairwise computation. A brief description of this application and its parallelization appears in the SPLASH report[SWG92], and [SHG92] is a more thorough study.

[3] `Mp3d` has since been rewritten at NASA-Ames using spatial decomposition techniques and has improved locality of reference[LLJ+92]. Unfortunately this improved code has not been publicly released.

Application	Access Rate	Reuse of Data	Comment
sieve	high	very high	regular access intervals
blkmat	low	low	varying intervals between accesses
sor	high	high	many barrier synchronizations
ugray	medium	medium	complex data structures
water	medium	medium	bursty traffic
locus	high	high	linear sequences of accesses
mp3d	high	low	changing data usage
barnes	medium	high	complex data structures

Table 2.2: Summary of Application Characteristics.

In this application, bodies are organized in a three dimensional hierarchical structure called an octree. This allows aggregation of distant particles for computational efficiency, but individual access to nearby particles for computational accuracy. This is a well crafted implementation that assigns neighboring particles to the same thread, and thus there is much reuse of data.

Hierarchical structures are used for both the organization of data and the partitioning of work among the threads. Building tree like structures can not be completely parallelized since there is little concurrency near the root of a tree. With large numbers of threads, a substantial amount of time is spent waiting for synchronization events. This is due both to incomplete parallelization of tree operations and to imperfect load balancing among the threads.

The main characteristics of `barnes` are:

- Moderate access rates.
- High reuse of data.
- Many long synchronization stalls.

2.2.9 Summary of Application Characteristics

Table 2.2 summarizes the preceding application discussions in terms of the applications' access rates and reuse of data. These characteristics will affect the results of the experiments presented in this dissertation. High access rates, for example, will require multithreading to use many threads per processor, and low reuse of data will cause caching to perform poorly.

2.3 Simulation System

In order to conduct this research we have built a fast and accurate simulator called FAST (for Fast Accurate Simulation Tool). In this section we summarize a few important details and then discuss the usage of the simulator for the studies conducted as part of this thesis. Appendix B contains a detailed discussion of the simulator and the techniques and tradeoffs chosen in its design.

2.3.1 The Simulator

The simulator is based on the technique of execution driven simulation. This is a process whereby the application program to be simulated is actually directly executed, but it has been modified so that it counts its own execution time and returns control to the simulator at special events, such as shared memory references. The simulator works by executing the many parallel threads for small periods of time, and then scheduling the resulting events so that they are all simulated in a correct global time order.

The modifications to the application program are best made at the object code level, since at this level accurate timing can be determined based on the individual assembly language instructions. All of our applications were compiled at optimization level "-O2", and their timing results are based on this.

The simulator accurately models the timing of the MIPS R3000[Kan89] pipeline, and all interactions between threads are accurately ordered. One slight inaccuracy occurs for simulations using caching of shared data: the cache interactions, such as invalidations, are done instantaneously rather than being delayed for the transit time for the invalidation messages to travel from the directory to the cache. This simplification makes the cache simulator much more efficient and easier to write, but means that data gets invalidated slightly sooner than it would on a real machine.

Because of careful use of execution driven simulation techniques, our simulator is approximately 50 times faster than comparable simulators such as Tango[DGH91] or [O'K89]. The main advantage of this speed is that it allows us to run longer and larger simulations (and thus more representative of large systems) than those of previous researchers.

Experiment: efficiency on an ideal machine			
Application	Processors	Multithreading	• Latency = 0 cycles
sieve	1–1024	1	
blkmat	1–1024	1	
sor	1–1024	1	
ugray	1–512	1	
water	1–343	1	
locus	1–128	1	
mp3d	1–1024	1	
barnes	1–512	1	

Table 2.3: Experimental parameters for measuring the execution efficiencies on an ideal machine.

2.3.2 Simulation Constraints

There are a number of constraints that have kept us from running simulations as large as we would have liked. First, some of the applications (`water` and `locus`) had only moderate input sizes available. Second, despite our fast simulator, simulation is still time consuming. We thus restricted problem sizes so that individual simulations completed within a few hours. And third, simulations of large parallel machines require a lot of space to hold the state of the many simulated threads and caches. We were limited to 128 mega-bytes that was available on the largest of our simulation host machines.

Table 2.1 listed the input sizes that were used for each of the applications. In order to gauge the amount of parallelism available, we have simulated the applications as if they were executing on an "ideal" machine that had 0 latency and no contention on accesses to shared memory. Such a machine would be impossible to build, but it corresponds to an upper bound on achievable performance. Table 2.3 list the experiment's parameters and Figure 2.7 shows the results.

Rather than show the standard speedup curves (speedup = execution time on 1 processor / execution time on P processors), we have plotted the efficiency vs. the number of processors (efficiency = speedup / P). Efficiency is much like processor utilization. The difference is that efficiency is directly related to performance, where as utilization is simply a metric of how busy the processors are. For example, processors might be busy spinning or doing redundant work and thus not contributing to overall speedup. The advantage of efficiency over speedup is that it has been normalized by the number of processors and

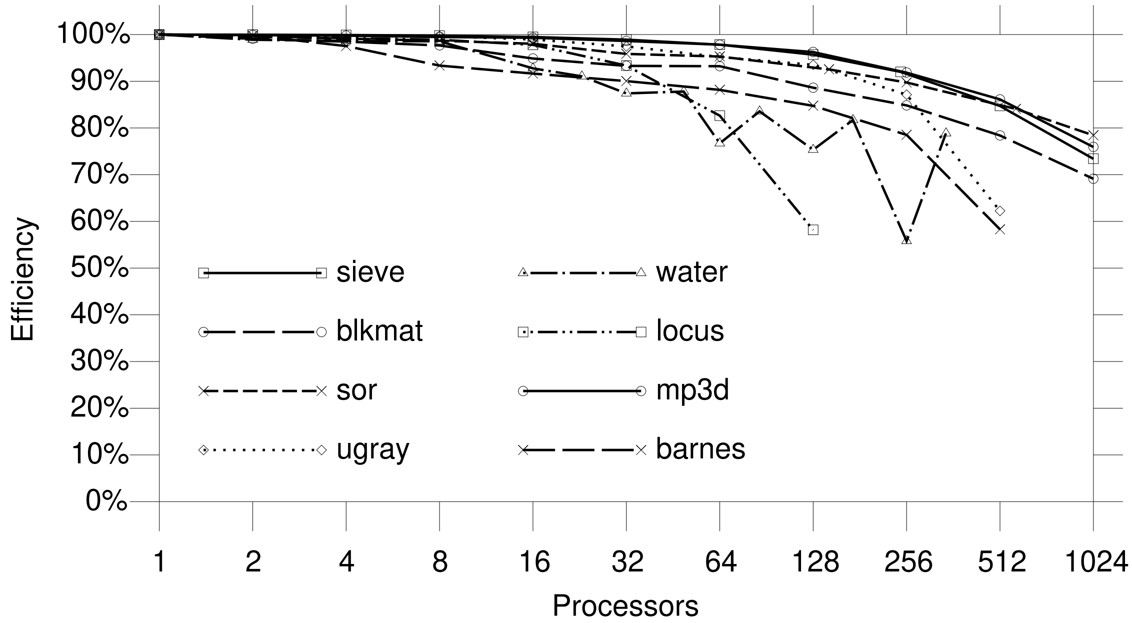


Figure 2.7: Efficiency on ideal (0 latency) machine.

thus can be used to compare machines of different sizes. As can be seen from Figure 2.7, efficiency is a fairly constant metric until the number of processors exceeds some limit, at which point it drops more quickly.

We are simulating fixed size problems, and thus as the number of processors is increased, the work gets partitioned more finely among a larger set of threads. The efficiencies degrade at some point because of various imperfections in the parallelization such as: uneven load balancing, synchronization overhead, redundant calculations, and occasionally restricted parallelism (such as at the root of a tree). `Water` stands out in Figure 2.7 because of its jagged efficiency curve. This is the result of poor load balancing that occurs when the number of processors is incongruent to the number of molecules (343), as was explained in Section 2.2.5.

Figure 2.7 was used to choose a “reasonable” limit on the number of threads (amount of parallelism) that could be used for each application given the fixed problem sizes that we were able to simulate. These thread limits are shown in Table 2.4. For the various simulation experiments in this thesis, these thread limits constrained the number of threads and processors that were used.

Thread Limits	
sieve	256
blkmat	256
sor	256
ugray	256
water	120
locus	64
mp3d	256
barnes	128

Table 2.4: Limits on the number of threads used from the various applications because of increasing performance degradation when trying to extract greater parallelism from fixed size problems.

These thread limits are not strict, and occasionally we exceed them. The main point is that with fixed problems sizes there is also a limit on the amount of parallelism available from the applications. If these applications are to be used on larger parallel machines, larger problem sizes will be needed to supply additional parallelism.

2.3.3 Revised Machine Model

In Section 2.1.1 we selected a network latency of 200 cycles based on our expectations of the latency commensurate with a 1000 processor machine. However because of the simulation constraints just discussed, we are unable to run problems large enough for such a machine. We have therefore reduced the number of processors, but kept the same network latency. Our revised machine model is depicted in Figure 2.8. Our typical simulation will be performed with from 16 to 32 processors.

We feel this reduction of the number of processors but maintaining the latency makes our results more directly applicable to a 1000 processor machine. One of the main objectives of this research is evaluating multithreading as a means of tolerating the long memory latencies of large machines. It will turn out that the main factor determining the efficacy of multithreading is the behavior of the individual threads. If threads execute for relatively long intervals between remote references, and if these intervals are fairly constant in length, then multithreading will work well. The “interval” behavior of these threads is primarily determined by the sequences of instructions in the application codes themselves, and not by the number of parallel threads or the size of the problems. Thus with larger

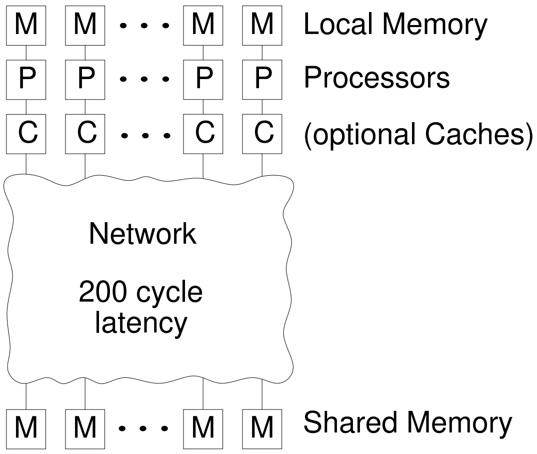


Figure 2.8: Revised model of parallel machine with fewer processors but the same latency as expected on a 1000 processor machine.

problems, we expect to see the same multithreading behavior on large machines as we observe in our studies of the reduced machine model.

A final note on our simulations is that all results are based solely on the parallel phase of computation. All of the applications studied in this dissertation have a sequential initialization phase, a parallel computation phase, and a sequential termination phase. It is common practice in simulation studies to report only the parallel phase. This is done for a number of reasons. First, many of the application are iterative, but only a small number of iterations is simulated, thus artificially decreasing the duration of the parallel phase and thereby increasing the significance of the sequential phases. Second, as problem sizes are increased, the sequential phases become a smaller and smaller fraction of the total computation[Gus88]. Third, many of these applications were written for today's small shared memory machines, and often much of the initialization could have been parallelized, but this was not deemed necessary on a small machine. Finally, a large part of the initialization and termination phases is input and output, which we expect to be done in parallel for larger machines.

Chapter 3

Behavior of Multithreading

In this chapter we present an abstract model of multithreaded systems. Its analysis provides intuition into the behavior that we will later observe in our simulation studies.

3.1 Multithreading Model

Our model of a multithreaded processor was formulated by Saavedra-Barrera, Culler, and von Eicken in [SBCvE90]. Following is restatement of the model and its basic analysis using the notation of this dissertation.

The model considers just a single processor of a parallel computer. We assume that a thread on this processor repeatedly issues remote references at an interval of R cycles (called the run-length), and that after issuing a remote reference, the thread must wait for the response to return before resuming execution. The response time depends upon the delay through the interconnection networks to and from memory, and for the model we assume a fixed round-trip latency of L cycles.

These two parameters, R and L, are shown in Figure 3.1. In this figure there is only a single thread executing on the processor, and the processor will have a utilization factor of $R/(R+L)$. For large shared memory parallel computers we expect L to be substantially larger than R, as shown in the figure, and thus the processor will be poorly utilized.

Figure 3.2 shows the same situation, but with three threads per processor. The number of threads per processor will be called M for multithreading level. Between threads we have added a context switch cost of C cycles, which we expect will be just a few cycles or even zero cycles with appropriate hardware support.

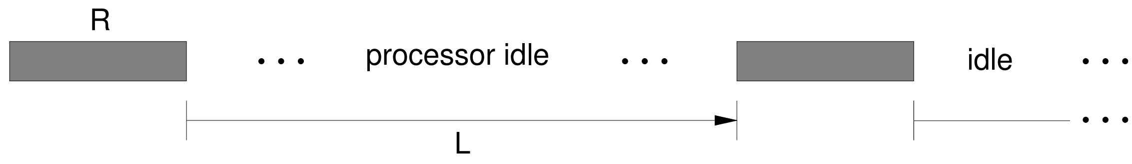


Figure 3.1: Model of a single thread. (R = Run-length, L = Latency)

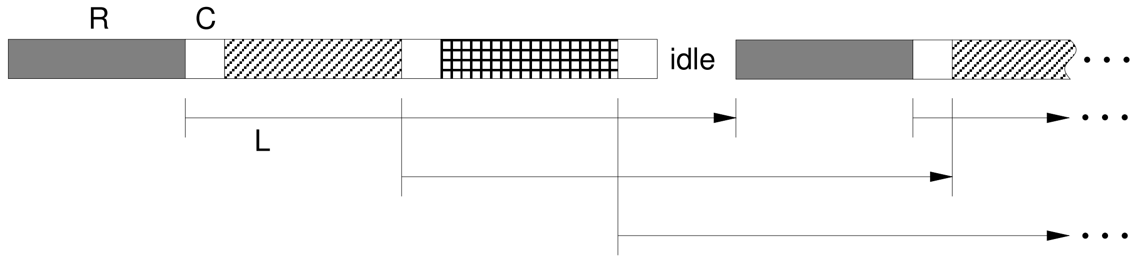


Figure 3.2: Multithreading with 3 threads per processor. (C = Context switch cost)

3.1.1 Analysis Under Constant Run-Lengths

Under simple assumptions about R, L, and C, we can compute the processor utilization as a function of the multithreading level M. The simplest assumption is that R, L, and C are all constants. Under this assumption, the performance analysis can be broken into two separate cases. The first case occurs when there are not enough threads to hide the latency, and thus the processor is sometimes idle. Figures 3.1 & 3.2 both show this case. The system forms a renewal process[Wol89] that starts at the end of each idle period. The length of the renewal period is $R + L$ and the amount of work done is $M \cdot R$. Thus processor utilization is $MR/(R + L)$. In this case performance increases linearly with the multithreading level. The second case occurs when there *are* enough threads to hide the latency. In this case the only performance loss comes from the context switch overhead, and thus processor utilization is $R/(R + C)$. The boundary between the two case occurs when $M(R + C) = R + L$. Solving for M we get:

$$\text{Processor Utilization} = \begin{cases} \frac{MR}{R+L} & \text{if } M < 1 + \frac{L-C}{R+C} \\ \frac{R}{R+C} & \text{otherwise} \end{cases}$$

If C is small, we can approximate the number of threads needed to maximize

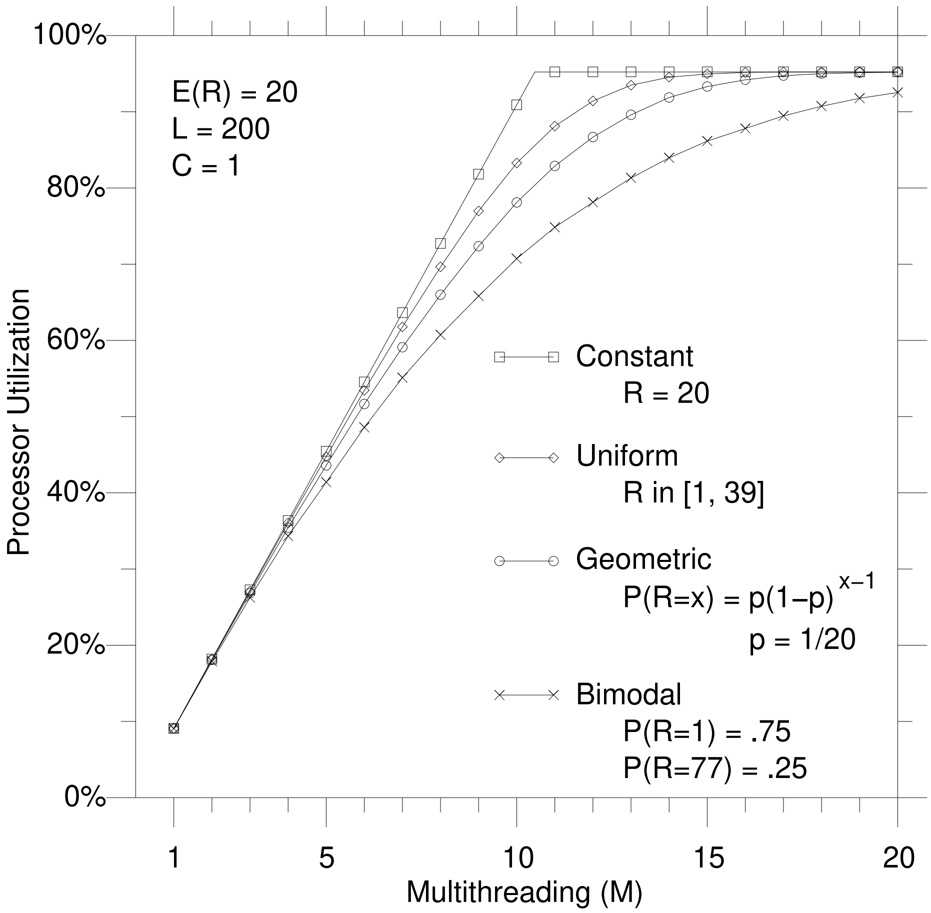


Figure 3.3: Processor utilization as a function of multithreading for various run-length distributions.

processor utilization as $M = 1 + L/R$.

Rule 1 *With a constant run-length distribution, approximately* $M = 1 + L/R$ *threads are needed to keep the processor busy.*

This function is shown graphically in Figure 3.3 as the curve labeled "constant", using the values of $R = 20$, $L = 200$, and $C = 1$, which are similar to the values that we expect for real applications and hardware.

3.1.2 More Complex Distributions

In real applications the run-lengths will rarely be predictable. A better model is to choose the run-lengths from some random distribution. Figure 3.3 also shows the processor utilization curves for three other distributions: uniform, geometric, and bimodal. Each of these is parametrized to have the same mean run-length ($R = 20$). For the uniform distribution, the run-lengths are chosen with equal probability over the range 1 to 39. For the geometric distribution, the run-length comes from a sequence of biased coin flips where at each step the probability of completion is $P = 1/20$. And for the bimodal distribution, the run-length is either: $R = 1$ with 75% probability, or $R = 77$ with 25% probability.

For the geometric run-length distribution the model was solved by Saavedra-Barrera and Culler[SBC91] using Markov chain analysis. However for more general distributions, analytic solutions become intractable. We have extended the applicability of the model by using numeric simulation to compute the processor utilization versus multithreading curves for any specified distribution. For the uniform and bimodal distributions, the curves in Figure 3.3 were calculated using this technique.

The histograms of these distribution functions are shown in Figure 3.4. These histograms are drawn with triangular piles who's area represents the percentage of total run-lengths having a particular value. Piles that would overlap are combined to make a larger pile. Refer to appendix A for a complete explanation and rationale of this new and somewhat odd histogram. These histograms will be used as a basis for building intuition into multithreading behavior.

By comparing the distributions and the plots of their performance, we can observe that the distributions with the most short run-lengths need the highest multithreading levels. This occurs even though the mean run-length remains the same (20) for all of these distributions. The short run-lengths cause problems when, by random chance, several threads have short run-lengths in succession, and the remaining threads are unable to hide the latency. In these cases the processor will be forced to stall. In the opposite case, when several successive threads all have long run-lengths, the latency is easily covered, but the excess latency tolerance is wasted.

Rule 2 *When run-lengths are random, the presence of short run-lengths increases the multithreading level needed to keep the processor busy.*

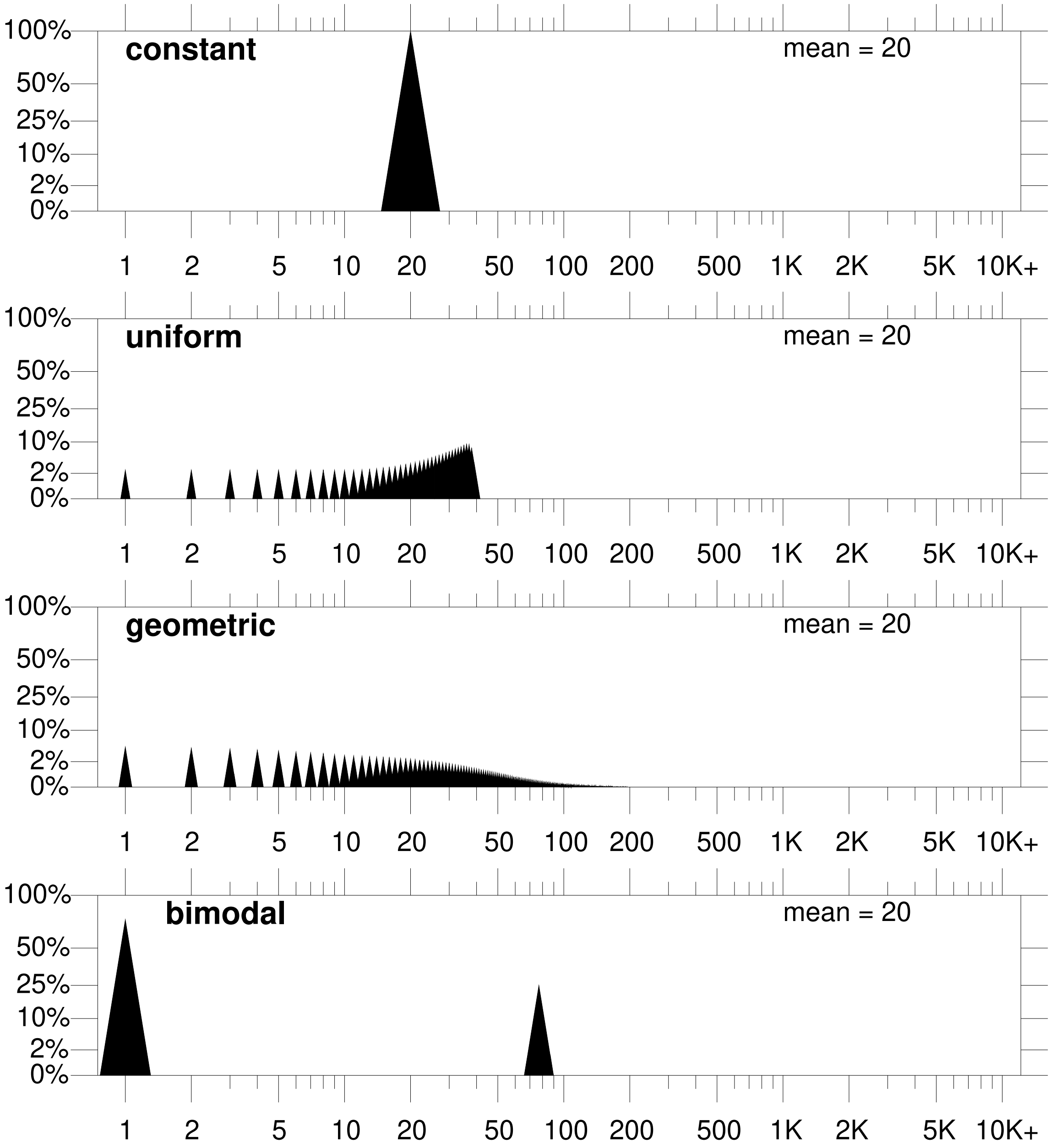


Figure 3.4: Histograms of distribution functions. The horizontal axis shows the run-length. Each data point is represented by a pile who's size corresponds to its percentage of the total, and overlapping piles combine together to make taller piles.

Experiment: run-lengths under switch-on-load			
Application	Processors	Multithreading	• Latency = 200 cycles • Context switch = 0 cycles • Scheduling = round robin • No shared memory caches
sieve	16	12	
blkmat	32	4	
sor	8	45	
ugray	8	16	
water	10	12	
locus	2	32	
mp3d	8	28	
barnes	12	13	

Table 3.1: Experimental parameters for run-lengths under **switch-on-load**.

3.2 Applications' Run-Length Distributions

Figures 3.5 & 3.6 show the run-length distributions for the benchmark applications under the **switch-on-load** multithreading model. These run-length distributions were collected from simulations as specified in Table 3.1. The performance results and simulation assumptions will be discussed in the next section. We discuss the run-length distributions first so that we can make predictions based on what we have learned from the multithreading model.

The applications have widely different run-length distribution functions and means. The first four applications are in fact similar to the four mathematical distributions given in the previous chapter: `sieve` is similar to the constant distribution, `blkmat` (excluding the pile near 20) is similar to the uniform distribution, `sor` is similar to the bimodal distribution, and `ugray` is similar to the geometric distribution. Also note that `water` has some very long run-lengths (around 5000 cycles). These occur because this application was written so that it copies shared data values into local memory and then performs large calculations using only the local copies. The `blkmat` application also makes local copies of shared data, and because of this, it achieves the highest average run-length (164 cycles). `Water` (43) and `barnes` (42) also exhibit long mean run-lengths. `Ugray` (22) and `sieve` (19) have more moderate run-lengths. And `mp3d` (9.5), `locus` (7), and `sor` (6) have short mean run-lengths.

Based on the mean run-lengths and the distributions functions, we can make predictions about how these applications will perform on a multithreaded machine. Because of

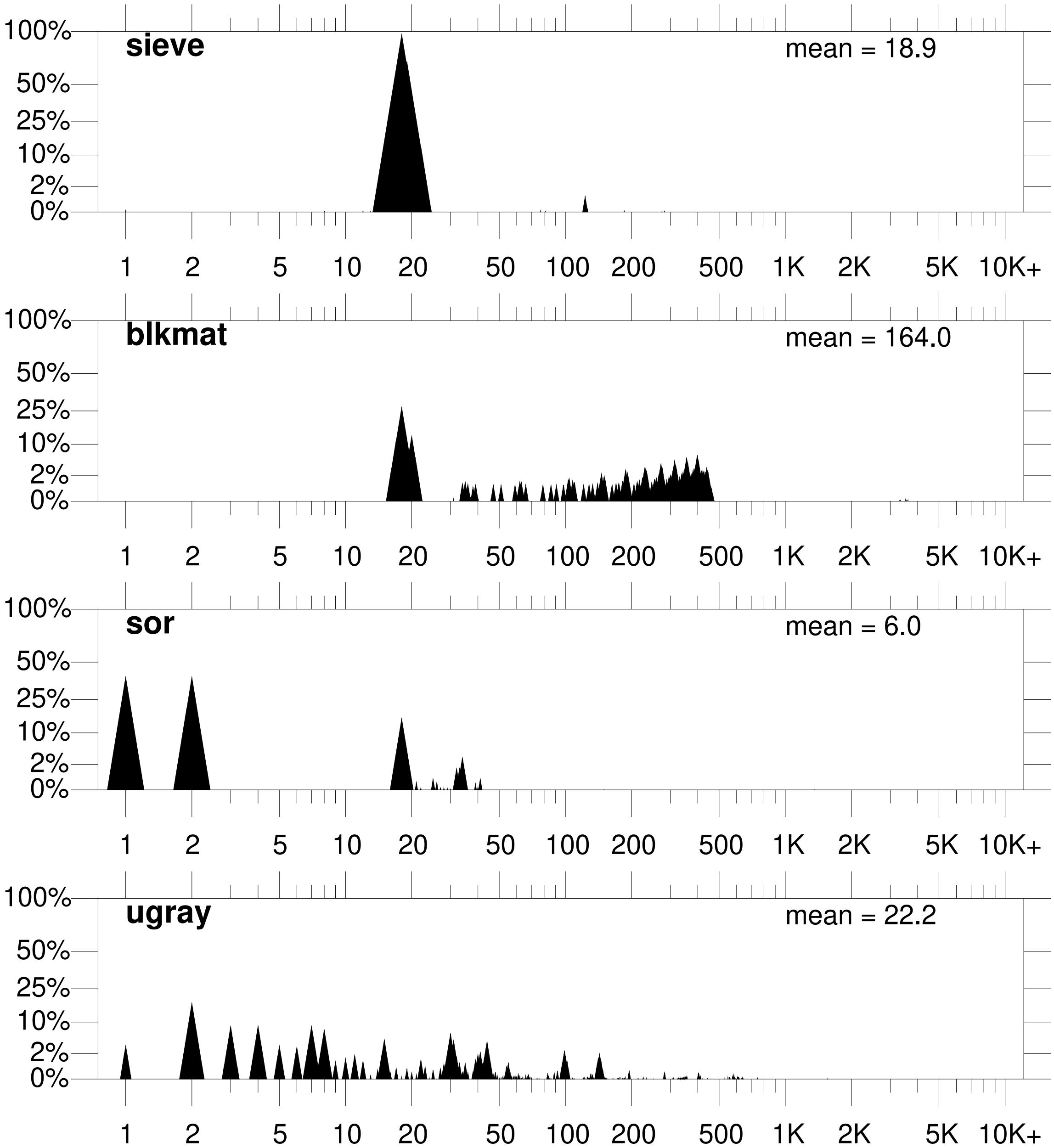


Figure 3.5: Histograms of the run-lengths distributions of the applications running under **switch-on-load**.