Run-Time Thread Management for Large-Scale Distributed-Memory Multiprocessors

by

Daniel Nussbaum

B.S., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1985

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1988

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1993

© Massachusetts Institute of Technology 1993

Signature of Author ....
Department of Electrical Engineering and Computer Science
August 31, 1993

Certified by ....
Anant Agarwal
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, EECS Committee on Graduate Students

Run-Time Thread Management for Large-Scale Distributed-Memory Multiprocessors

by

Daniel Nussbaum

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 1993, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Effective thread management is crucial to achieving good performance on large-scale distributed-memory multiprocessors that support dynamic threads. For a given parallel computation with some associated task graph, a thread-management algorithm produces a running schedule as output, subject to the precedence constraints imposed by the task graph and the constraints imposed by the interprocessor communications network. Optimal thread management is an NP-hard problem, even given full *a priori* knowledge of the entire task graph and assuming a highly simplified architecture abstraction. Thread management is even more difficult for dynamic data-dependent computations which must use online algorithms because their task graphs are not known *a priori*. This thesis investigates online thread-management algorithms and presents **XTM**, an online thread-management system for large-scale distributed-memory multiprocessors. **XTM** has been implemented for the MIT Alewife Multiprocessor. Simulation results indicate that **XTM**'s behavior is robust, even when run on very large machines.

XTM makes the thread-management problem more tractable by splitting it into three sub-problems:

1. determining what information is needed for good thread management, and how to efficiently collect and disseminate that information in a distributed environment,
2. determining how to use that information to match runnable threads with idle processors, and
3. determining what interprocessor communication style **XTM** should use.

XTM solves these sub-problems as follows:

1. Global information is collected and disseminated using an X-Tree data structure embedded in the communications network. Each node in the X-Tree contains a "presence bit," the value of which indicates whether there are any runnable threads in the sub-tree headed by that node. On a machine with a sufficiently high, balanced workload, the expected cost of maintaining these presence bits is proved to be asymptotically constant, regardless of machine size.
2. The presence bit information, along with a combining process aided by the X-Tree, is used to match threads to processors. This matching process is shown to be eight-competitive with an idealized adversary, for a two-dimensional mesh network.
3. A message-passing communication style yields fundamental improvements in efficiency over a shared-memory style. For the matching process, the advantage is shown to be a factor of $\log l$, where l is the distance between an idle processor and the nearest runnable thread.

Asymptotic analyses of **XTM**'s information distribution and thread distribution algorithms are given, showing **XTM** to be competitive with idealized adversaries. While the solutions to the sub-problems given above have provably good characteristics, it is difficult to say anything concrete about their behavior when combined into one coherent system. Simulation results are therefore used to confirm the validity of the analyses, with the Alewife Multiprocessor as the target machine.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Engineering

Acknowledgments[1]

I owe many people thanks and more for the assistance and support they lent during the time I was working on this thesis. I owe the most to Professor Anant Agarwal, my thesis advisor, for the guidance, encouragement and prodding he gave me throughout the course of the project. Many times I walked into his office convinced that this work was worthless and that the thesis would be a failure. Every time, I walked out armed with a new outlook and new and fresh ideas about how to proceed. In the time I have worked with Anant, not only have I learned about runtime systems, I have also developed ways of looking at large and complex computer systems that will be applicable throughout my career. Anant has been a pleasure to work with and to learn from.

I also want to thank Steve Ward, Bert Halstead and Baruch Awerbuch for the advice and insights I got from them. Steve and Bert not only acted as thesis readers, they also got involved early enough in the process that they could offer constructive suggestions and criticisms throughout the project. Baruch gave invaluable advice on theoretical matters. He first suggested modifying **XTM** into **XTM-C**, which seemed to be easier to write proofs for. We are currently working together on polylog-competitiveness proofs for **TTM**, **XTM** and **XTM-C**.

David Kranz, Beng-Hong Lim, David Chaiken, Kirk Johnson, Patrick Sobalvarro and Terri Iuzzolino took the time to read early, unreadable versions of this thesis. If the resulting work bears any resemblance to a coherent whole, it is through their efforts. Any remaining errors of omission, commission, awkwardness or inaccuracy are entirely my fault.

John Kubiatowicz, my office-mate, has probably borne the brunt of this thesis. He has had to put up with all kinds of annoyances, ranging from two-hour phone calls to the hurling of insults and stacks of paper. My only saving grace is that I expect no different from him when he writes his thesis.

Anne McCarthy was responsible for insulating me from most of the administrivia that can drive you crazy when other things have already pushed you over the edge. She handled all the planning for committee meetings, thesis talks and the like, and she constantly offered her assistance in any way that would save me from getting tangled up in red tape. If she had written this thesis, it would have been finished years ago.

Finally, I would like to thank my family and friends for bearing with me for all these years. There were times when I was convinced I would never finish, but my parents always kept the faith. They, more than anybody, convinced me that the long haul was worth it, and that I should see this through to the end. My brother Michael, my sister Beth and my sister-in-law Jill were also full of encouragement, willing to offer a sympathetic ear or a warm meal when the whole thing seemed overwhelming.

I can only hope that my gratitude can repay all these people for the help and support they have given me. Without them, this thesis would never have been written.

[1]The research reported in this thesis was funded by ARPA grant # N00014-91-J-1698 and NSF grant # MIP-9012773.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

MIMD multiprocessors can attack a wide variety of difficult computational problems in an efficient and flexible manner. One programming model commonly supported by MIMD multiprocessors is the *dynamic threads* model. In this model, sequential *threads of execution* cooperate in solving the problem at hand. Thread models can be either *static* or *dynamic*. In the *static threads* model, the pattern of thread creation and termination is known before run-time. This makes it possible for decisions regarding placement and scheduling of the threads to be made in advance, either by the user or by a compiler. Unfortunately, many parallel applications do not have a structure that can be easily pre-analyzed. Some have running characteristics that are data-dependent; others are simply too complex to be amenable to compile-time analysis. This is where the dynamic aspect of the *dynamic threads* model enters the picture. If such programs are to be run in an efficient manner on large-scale multiprocessors, then efficient run-time thread placement and scheduling techniques are needed. This thesis examines the problems faced by an on-line thread-management system and presents **XTM**, an **X**-Tree-based [25, 6] **T**hread-**M**anagement system that attempts to overcome these problems.

The general thread-management problem is NP-hard [7]. The standard problem has the following characteristics: precedence relations are considered, the communications network is flat and infinitely fast, tasks take differing amounts of time to finish, preemptive scheduling is not allowed, and the thread-management algorithm can be sequential and off-line. Even if precedence relations are ignored, the problem is still NP-hard when tasks vary in length and preemption is not allowed: it reduces to the bin-packing problem [7]. In the real world, such simplifications often do not apply: real programs contain precedence relations, communications networks are neither flat nor infinitely fast, and in order for thread management algorithms to be useful, they must be distributed and run in real time.

Since the overall problem is too difficult to tackle all at once, **XTM** breaks the thread-management problem down into three sub-problems, attacking each one separately. The sub-problems are identified as follows:

1. determining what global information is needed for good thread management and how to efficiently collect and disseminate that information in a distributed environment,

2. determining how to use that information to match runnable threads with idle processors, and

3. determining what interprocessor communication style to use.

For each of these sub-problems, we present a solution and show through formal analysis that the chosen solution has good behavior. Finally, we demonstrate, using high-level simulation results, that the mechanisms work well together.

1.1 Principles for Algorithm Design in Distributed Environments

The optimal thread management problem is NP-hard, even when significantly simplified. The problem is further complicated by the requirement that it be solved in a distributed fashion. However, it is neither necessary nor practical to expect a multiprocessor thread manager to achieve an *optimal* schedule. The primary goal of such a system is to maximize processor utilization, thus minimizing overall running time. This task is especially challenging because information about the state of the machine is generated locally at the processors, but information about the state of the entire machine is needed in order to make good thread-management decisions. Collection and distribution of this type of global state information is difficult in a distributed environment. Therefore, we set forth several general principles to guide our design efforts:

Eliminate ALL Hot-Spots: At any given time, the number of processors accessing a single data object and the number of messages being sent to a single processor should be limited. Otherwise serialization may result, with a corresponding loss in efficiency.

Preserve Communication Locality: Threads should be run physically near to the data they access, so as to minimize time spent waiting for transmissions across the communication network.

Minimize System Overhead: Overhead imposed by the thread manager should be kept to a minimum: time spent by a processor running system management code is time spent not running the application being managed.

Given a design choice, the path that follows these principles more closely will be more likely to attain good performance in a large-scale distributed system. Experience shows that overall system performance will suffer if any piece of the thread-management system should fail to follow any of these principles.

1.2 Contributions of This Thesis

This thesis examines and develops thread-management algorithms for large-scale distributed-memory multiprocessors. Guided by the design principles given above, we have developed **XTM**, a thread management system that is sound from both a theoretical and a practical perspective.

XTM solves the sub-problems identified above as follows:

1. Global information is collected and disseminated using an X-Tree [25, 6] data structure embedded in the communications network (see Figures 4-1 and 4-2). Each node in the tree contains a "presence bit" whose value indicates whether there are any runnable threads in the sub-tree headed by that node. We show that on a machine with a sufficiently high, balanced workload, the expected cost of maintaining these presence bits is asymptotically constant, regardless of machine size.

Presence-bit maintenance follows a simple `or` rule at the nodes of the tree. The state of a node's presence bit only changes when its *first* child's presence bit goes from zero to one, or when its *last* child's presence bit goes from one to zero. In this fashion, presence bit update messages are naturally combined at the nodes of the tree. This combining behavior has the effect of avoiding hot-spots that may otherwise appear at higher nodes in the tree and reduces **XTM**'s bandwidth requirements. Furthermore, the tree is embedded in the communications network in a locality-preserving fashion, thus preserving communication locality inherent to the application. Finally, the operations involved in maintaining global information are very simple, burdening the system with very little overhead.

2. The presence bit information, along with a combining process aided by the X-Tree, is used to match threads to processors. We show that this matching process can take no more than eight times as much time as a particular idealized (unimplementable) adversary, running on a two-dimensional mesh network.

   Multiple requests for work from one area of the machine are combined at the X-Tree nodes, allowing single requests for work to serve many requesters. In this manner, large volumes of long-distance communication are avoided, and communication locality is enhanced. Furthermore, if a single area of the machine contains a disproportionately large amount of work, a few requests into that area are made to serve large numbers of requesters, therefore avoiding hot-spot behavior in that area of the machine.

3. A message-passing communication style yields fundamental improvements in efficiency over a shared-memory style. For the matching process, the advantage is a factor of $\log l$, where l is the distance between an idle processor and the nearest runnable thread.

   The message-passing vs. shared-memory design choice boils down to an issue of locality. In a message-passing system, the locus of control follows the data; in a shared-memory system, the locus of control stays in one place. The locality gains inherent to the message-passing model yield significant performance gains that appear in both analytical and empirical results.

Chapter 5 gives asymptotic analyses of **XTM**'s information-distribution and thread-distribution algorithms that show **XTM** to be competitive with one idealized adversary.

In summary, **XTM** makes use of an X-Tree data structure to match idle processors with runnable threads. The X-Tree is used to guide efficient information distribution about where runnable threads can be found. The tree is also used to *combine* requests for work, so that a single work request can bring work to multiple idle processors. Finally, the mapping of the tree onto the physical processor array enhances communication locality of the application in a natural way, usually causing threads to run on processors near to where they were created.

An implementation of **XTM** has been written for the MIT Alewife Multiprocessor [1, 2]. Alewife provides both an efficient implementation of the shared-memory abstraction and efficient interprocessor messages. **XTM** employs message-passing as its primary communication style. This message-passing implementation is made possible by the static mapping of the X-Tree data structure onto the physical processor array. Use of messages not only lowers the cost of primitive functions like thread creation, but it also improves communication locality over a shared-memory implementation. These locality-related gains are shown to become important as machine size increases. This thesis presents a detailed description of **XTM**. It presents asymptotic analyses of **XTM**'s information distribution and thread-distribution algorithms, showing **XTM** to be competitive with idealized algorithms. Simulation results bear out the theoretical analyses.

In the process of studying the behavior of **XTM** and other thread-management algorithms, we have come to the following conclusions, using both analytical and empirical arguments:

- As machines become large (≥ 256 processors), communication locality attains a position of overriding importance. This has two consequences: First, a thread manager is itself an application. If the structure of the application is understood well enough, it can be implemented in a message-passing style, instead of using shared-memory. A message-passing implementation can achieve significant performance gains by lowering demands on the communication system. This is the case when the locus of computation follows the data being manipulated, drastically reducing the cost of accessing that data. Second, a good thread manager should attempt to keep threads that communicate with one another close together on the machine.

- Thread management is a global optimization problem. A good thread manager must therefore achieve efficient collection and distribution of relevant global information.

- Parallel algorithms for large machines must avoid hot-spot behavior, or else risk losing the benefits of large-scale parallelism. Therefore, the algorithms presented in this thesis are all fully distributed, employing *combining* techniques for the collection and distribution of the threads being managed and of the global information needed to achieve good thread management.

While the solutions to the sub-problems given above have provably good characteristics, it is difficult to say anything concrete about their behavior when combined into one coherent system. In order to study the behavior of different thread-management algorithms and to confirm the validity of the analyses, we developed a simulator for large-scale multiprocessors. This simulator, called PISCES, models the Alewife architecture and produced nearly all of the data comparing different thread managers on various-sized multiprocessors. These simulation results confirmed the theoretical results: for large machines, the techniques employed by **XTM** performed well. For example, a numerical integration application run on 16384 processors and managed by **XTM** ran ten times faster than the same application managed by a diffusion-based thread manager, and three times faster than the same application managed by a round-robin thread manager. In fact, the **XTM** run was within a factor of three of a tight lower bound on the fastest possible running time.

1.3 Systems Framework and Assumptions

The systems framework under which this research was performed has the following characteristics:

1. Thread creation is fully dynamic: a new thread may be created on any processor at any time.

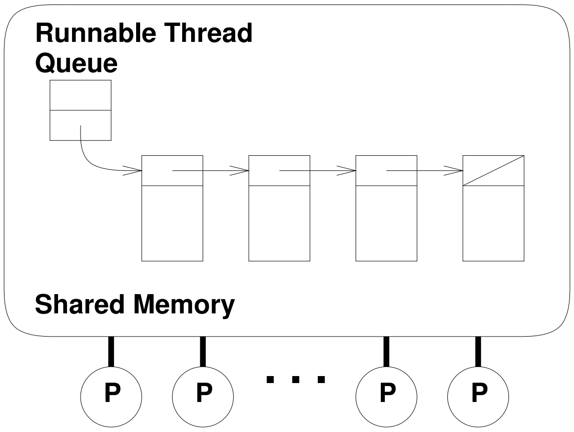


Figure 1-1: Self-Scheduled Model: *Any thread can run on any processor. The processors themselves determine which thread runs on which processor.*

2. Threads follow the Self-Scheduled Model [14], as depicted in Figure 1-1. This means that the processors that execute application code also contend between themselves to decide on which processor the various threads that make up the application are run. There is no fundamental link between threads and processors: any thread can run on any processor. The actual thread queue is implemented in a distributed fashion. Every processor maintains its own local queue of threads (see Figure 1-2).

3. The scheduling policy is non-preemptive. A processor runs a thread until the thread either terminates or blocks on some synchronization construct. When the thread running on a processor blocks or terminates, the processor becomes *idle*: it needs to find another runnable thread to execute. The behavior of an idle processor varies significantly depending on the thread-management strategy. When a consumer-driven search is in place, idle processors search the machine for work; when a producer-driven search is being used, idle processors simply wait for more work to be delivered to them.

   This non-preemptive policy places two requirements on applications being managed. First, fairness among threads that constitute an application must not be an issue. Second, deadlock issues are handled entirely by the application. We assume that applications that need a contended resource will block on a synchronization construct associated with the resource, thus taking themselves out of the runnable task pool.

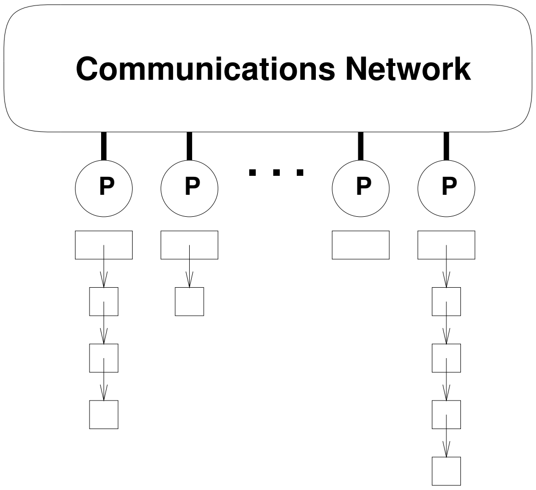


Figure 1-2: Implementation of the Self-Scheduled Model: *Every processor maintains a local queue of threads.*

4. Differences between individual threads are ignored. Threads can differ with respect to such parameters as running time and communication behavior. For the purposes of this thesis, we assume that all threads run for the same amount of time.

5. For the purposes of formal analysis, we assume that there is no inter-thread communication. Furthermore, the shape of the application's task graph is assumed to be unknown. However, the actual implementation of **XTM** optimizes for the case where the task graph is a tree, with the likelihood of inter-thread communication between two threads diminishing with distance in the tree.

In order to analyze the performance of any algorithm on any multiprocessor, we need to know the communication structure provided by the machine. In this thesis, we examine that class of machines based on k-ary n-dimensional mesh communications networks connecting p processors, where $p = k^n$. Such an architecture is simple and can scale to arbitrarily large sizes without encountering wire-packing or wire-length problems for $n \leq 3$ [5]. Similar analyses can be performed for networks with richer communication structures. In our analyses, we ignore the effects that network contention may have on performance.[1]

[1] In [10], it is shown that for a large class of machines, the effect of network contention on performance is no worse than the effect of network latency, within a constant factor. This is true for all of the machines and applications simulated for this thesis.

Parameter	Description
p	Number of processors on a particular machine
P_i	Labeling for processor number i
Parameters for a k-ary n-cube Mesh Network	
k	Network Radix
n	Network Dimensionality
$P_{i_0,i_1,\ldots,i_{n-1}}$	Labeling for processor with mesh coordinates $i_0, i_1, \ldots i_{n-1}$: $0 \leq i_x \leq k-1$
t_n	Network Speed: Cycles per Flit-Hop

Table 1.1: Machine Parameter Notation.

1.4 Terminology and Notation

Table 1.1 lists notation used throughout this thesis. The parameters in Table 1.1 describe the particular machine under discussion, in terms of size, labeling, network parameters and the ratio between network speed and processor speed. A multiprocessor of a given size is referred to as a p-processor machine; the processors on that machine are labeled P_i, where i varies from 0 to $p-1$. Mesh-based multiprocessors are defined in terms of n, their *dimensionality*, and k, their *radix*. On a k-ary n-cube mesh multiprocessor, $p = k^n$. Finally, the ratio between processor speed and network speed is given as t_n, the number of processor cycles it takes for one flit to travel one hop in the interconnection network.

1.5 Outline

The rest of this thesis is organized as follows. Chapter 2 presents other research in this area. Chapter 3 discusses high-level decisions that were made in the early stages of **XTM**'s design. In Chapter 4, we give a more detailed presentation of the information-distribution and thread-distribution algorithms at the heart of **XTM**. These algorithms are subject to a formal analysis in Chapter 5, especially with respect to asymptotic behavior. An empirical approach is taken in Chapters 6 and 7. Chapter 6 describes the experimentation methodology used in examining the behavior of **XTM**'s algorithms and other interesting thread-distribution algorithms; Chapter 7 compares **XTM** with the other thread-management algorithms, as run on several dynamic applications. Finally, Chapter 8 presents conclusions and suggestions for future research.

Chapter 2

Background

Although the practical aspects of thread management and load sharing have been discussed in the literature for the past decade, thread management for large-scale multiprocessors has only been seriously examined during the latter half of that period. Kremien and Kramer [13] state

> ...a flexible load sharing algorithm is required to be general, adaptable, stable, scalable, transparent to the application, fault tolerant and induce minimum overhead on the system...

This chapter explores the work of other investigators in this area and evaluates that work with respect to Kremien and Kramer's requirements.

Znati, *et. al.* [31], give a taxonomy of load sharing algorithms, a modified version of which appears in Figure 2-1. Characteristics of generalized scheduling strategies are discussed in [16], in which the scalability of various candidate load-sharing schemes is examined, and [30], which looks at the effects of processor clustering.

In order to meet the criteria given in [13], a thread management system must be dynamic and fully distributed. In this thesis, we are especially interested in dynamic methods because we want to be able to solve problems whose structure is not known at compile time. The first widely-discussed dynamic Self-Scheduling schemes were not fully distributed; they employed a central queue, which posed no problem on small machines.

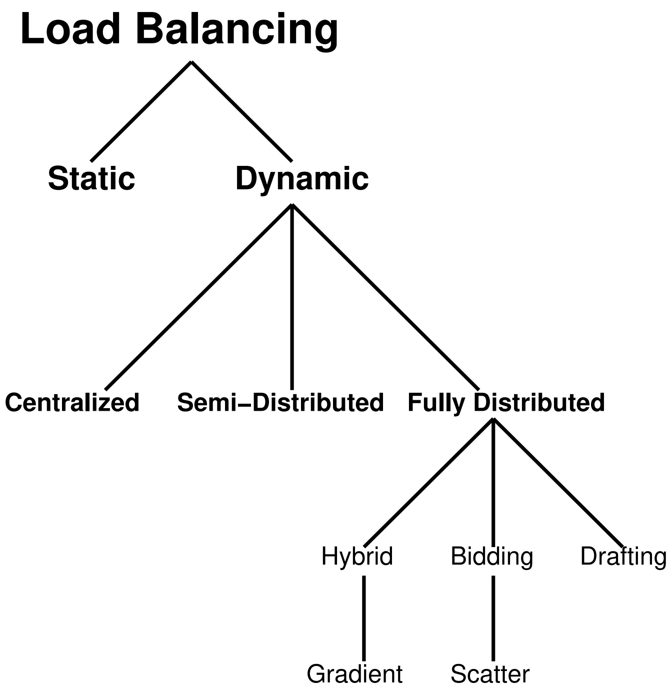


Figure 2-1: Taxonomy of Load Sharing Methods.

2.1 Centralized Self-Scheduling

One of the first references to the Self-Scheduling model in the literature appeared in 1985 in [14]. In this paper, the authors described Self-Scheduling, although they never explicitly named it such:

> There are p processors, initially idle. At $t = 0$, they each take K subtasks from a job queue, each experiencing a delay h in that access. They then continue to run independently, taking batches of jobs and working them to completion, until all the jobs are done.

Assuming that the subtask running times are independent identically distributed random variables with mean μ and variance σ^2, and given a requirement that K remain constant, the authors derive an optimal value for K, as a function of n (the total number of subtasks), h, p, and σ. They go on to show that when $\frac{n}{p \log p} \geq 1$ and $\frac{\sigma}{\mu} \leq 1$, the system efficiency[1] is

[1] System efficiency, for a given problem run on a given number of processors, is defined to be the total number of cycles of useful work performed divided by the total running time (in cycles) times the number

at least $\frac{1}{1+\sqrt{2}}$.

In 1987, Polychronopoulos and Kuck introduced Guided Self-Scheduling [21]. Their scheme, which is oriented towards the efficient scheduling of parallel FORTRAN loops, varies K with time. More specifically, whenever a processor becomes idle, it takes $\lceil \frac{r}{p} \rceil$ jobs from the central queue, where r is the number of jobs in the queue at the time. Guided Self-Scheduling handles wide variations in thread run-times without introducing unacceptable levels of synchronization overhead. For certain types of loops, they show analytically that Guided Self-Scheduling uses minimal overhead and achieves optimal schedules.

All of the early Self-Scheduling work assumes that h is both independent of p and unaffected by contention for the central job queue. These assumptions limit this work to be only applicable to small machines.

2.2 Fully Distributed On-Line Algorithms

We define fully distributed algorithms to be algorithms that contain no single (or small number of) serialization points on an arbitrarily large multiprocessor. Znati, *et. al.* [31], divide such algorithms into three sub-categories: bidding methods, drafting methods and hybrid methods. In bidding or producer-oriented methods, the creators of new work push the work off onto processors with lighter loads. In drafting or consumer-oriented methods, processors with light workloads locate and then steal excess work from processors with heavier workloads. In hybrid methods, producers and consumers cooperate in the load-sharing process. In the rest of this section, we give examples that have appeared in the literature for each of these categories of load-sharing algorithms.

2.2.1 Bidding Methods

Wu and Shu [28] present a Scatter Scheduling algorithm that is essentially the producer-oriented dual of the round-robin drafting algorithm described in Chapter 6. When running

---

of processors the problem was run on:

$$E = \frac{W}{pT}$$

this algorithm, processor i sends the first thread it creates to processor $i+1$, the second to processor $i+2$, and so on. Assuming that the application is suitably partitioned, this algorithm should get a relatively even load balance on machines of small and moderate size. This expectation is borne out in the results presented in [28]. However, since producer-processors send threads to essentially arbitrary destinations over time, any aspect of locality concerning data shared between related threads is lost. Furthermore the cost of thread creation goes up with the diameter of the machine, so on large machines, one would expect this algorithm to behave rather poorly.

Znati, *et. al.* [31], present a bidding scheme that takes distance between sender and receiver into account. When a new thread is created on a given processor, the processor recomputes and broadcasts its load to the rest of the system. Every processor then compares this new load with its own, taking the distance between itself and the source processor and itself into account. All processors that are eligible to receive the task based on the source processor's load, their own load and the distance between source and potential destination then contend for the job, the winner being the closest processor with the lightest load.

There are two problems with this scheme, both related to scalability. First, every time a new task is created, a global broadcast takes place. The bandwidth requirement for such a scheme is not satisfiable on an arbitrarily large machine. Second, every processor has to participate in every scheduling decision. As the multiprocessor gets large and the corresponding number of threads needed to keep it busy gets large, the thread scheduling overhead required of each processor will become unacceptably large. In fairness to the authors, one must realize that this work was done in the context of

> ...a loosely coupled large scale multiprocessing system with a number of processing elements interconnected through a broadcast based communication subnet...

This statement implies that the machines this algorithm is intended for are of limited size (despite the use of "large-scale"), and that the grain size of the tasks is also rather large, minimizing the effect of scheduling overhead on overall performance.

2.2.2 Drafting Methods

Ni, Xu and Gendreau [20] present a drafting-style load-sharing algorithm that also seems to be oriented towards a relatively small number of independent computers connected to a single Local Area Network. In this scheme, each processor has an idea of the state of all other processors by maintaining a "*load table*," which has an entry for every other processor in the system. Processors can be in one of three states, light load, normal load or heavy load. As in [31], every processor in the system is informed of all changes in load on every other processor by means of broadcasts. When a processor P_i goes into the lightly loaded state, it sends a request for work to every processor in the system that it thinks is in the heavy load state. Each of those processors will respond with a "draft-age," which is a measure of how much the entire system would benefit from an exchange of work between that processor and P_i. P_i then determines which candidate will yield the highest benefit and gets work from that processor.

The same objections that applied to the scheme proposed in [31] apply here: such an algorithm isn't really scalable. Furthermore, the "draft-age" parameter used to compare drafting candidates does not take the distance between the two processors into account. This lack of attention to communication distances further limits this algorithms effectiveness on large multiprocessors.

2.2.3 Hybrid Methods

Diffusion

Halstead and Ward proposed diffusion scheduling [9] as a means of propagating threads throughout the machine. Their description is a rather brief part of a larger picture and gives no specific details. However, the settling time for Jacobi Relaxation is proportional to the square of the diameter of the mesh upon which the relaxation takes place [3]. More sophisticated relaxation techniques, such as Multigrid methods [3], achieve much better convergence times at the expense of the locality achieved by simple diffusion methods. The **XTM** algorithm presented by this thesis can be thought of as the multigrid version of diffusion scheduling. We implemented both the simple diffusion scheduling algorithm described here and **XTM**. Results are given in Chapter 7.

Gradient Model

The Gradient Model proposed in [19] has a communication structure similar to that of diffusion scheduling. Processors are defined to be in one of three states: lightly loaded, moderately loaded or heavily loaded. Instead of moving threads based on the difference in the workload on neighboring processors, this scheme builds a gradient surface derived from estimates of the distance to the nearest lightly loaded processor. The gradient surface is defined to have a value of zero on lightly loaded processors; on every other processor, the value of the surface is defined to be one more than the minimum of its neighbors. The resulting surface gives an indication of the distance from and direction towards the nearest lightly loaded processor. The gradient surface is built by propagating information between nearest neighbors. A heavily loaded processor acts as a job *source*, sending jobs down the gradient in any "downhill" direction. A lightly loaded processor acts as a job *sink*, accepting jobs flowing towards it. A moderately loaded processor accepts jobs as if it were lightly loaded, but acts like a heavily loaded processor with respect to building the gradient. Finally, when there are no lightly loaded processors in the system, the gradient surface eventually flattens out at a maximum value equal to the machine diameter plus one. A machine in this state is said to be *saturated.*

The Gradient Model appears to scale well, independent of machine topology. However, there are some questions regarding its performance. The first question concerns the behavior of a saturated machine. In such a case, when a single processor becomes lightly loaded, a wave propagates out from that processor throughout the entire machine. In this manner, a number of tasks proportional to the square of the diameter of the machine will move between processors in response to a small local perturbation in the state of the machine. Second, gradient construction and job propagation takes place as a periodic process on every processor regardless of the state of the machine. This imposes a constant overhead on all processors independent of how well the processing load is spread about the machine: a price is paid for load-sharing whether needed or not. Finally, there is some question as to the stability of this scheme. It is easy to see that it tends to move jobs from heavily loaded processors towards lightly loaded processors; however, it seems possible to construct situations in which the time lag imposed by the propagation of gradient information could cause jobs to pile up at a single location, leading to a poor load balance at some time in the future.

Random Scheduling

Rudolph, *et. al.* [24], propose a load balancing scheme in which processors periodically balance their workloads with randomly selected partners. The frequency of such a load balancing operation is inversely proportional to the length of a processor's queue, so that heavily loaded machines (which have little need for load balancing) spend less time on load balancing leaving more time for useful computation. A probabilistic performance analysis of this scheme shows that it tends to yield a good balance: all task queues are highly likely to be within a small constant factor of the average task queue length.

Unfortunately, this work assumes that the time to perform a load balancing operation between two processors is independent of machine size. Clearly, in a large machine, it is more costly to balance between processors that are distant from each other than between processors that are close to each other. Since this scheme picks processors at random, the cost of a load balancing operation should rise for larger machines. Also, nothing is said in this paper about settling times or rates of convergence. The claim that a load-balancing scheme yields a balanced system is not worth much if the time it takes to achieve that balance is longer than the time a typical application is expected to run.

We have briefly surveyed a number of thread management algorithms given in the literature. For each algorithm, we have listed one or more potential problems that may be encountered when implementing the algorithm on a large-scale multiprocessor. In the rest of this thesis, we describe and evaluate a new thread management scheme that attempts to overcome all of these objections and tries to meet the requirements set forth by Kremien and Kramer [13].

Chapter 3

High-Level System Design

In this chapter, we present several high-level decisions that we made early on in the **XTM** design effort. These decisions were based mainly on the design principles given in Chapter 1: eliminate hot-spots, preserve communication locality and minimize system overhead. The details of the algorithms used by **XTM** will be given in Chapter 4. The goal of this chapter is to give the intuition behind the design of those algorithms.

As stated in Chapter 1, we break the thread management problem down into three sub-problems:

1. determining what global information is needed for good thread management and how to efficiently collect and disseminate that information in a distributed environment,

2. determining how to use that information to match runnable threads with idle processors, and

3. determining what interprocessor communication style to use.

Stated briefly, the high-level solutions to each of those sub-problems are:

1. Global information is collected and disseminated using an X-Tree [25, 6] data structure embedded in the communications network. Each node in the tree contains a "presence bit" whose value indicates whether there are any runnable threads in the sub-tree headed by that node.

2. The presence bit information, along with a combining process aided by the X-Tree, is used to match threads to processors.

3. A message-passing communication style yields fundamental improvements in efficiency over a shared-memory style.

In this chapter, we discuss each of these solutions in detail.

3.1 Global Information Management

In order to make good scheduling decisions, information about the state of the entire machine is needed, but this information is generated locally at the processors. A tree is a scalable data structure that can be used to efficiently collect and disseminate global information while avoiding hot-spots through combining techniques. Therefore, we use a tree-based data structure to aid in the efficient distribution of information about the state of the machine.

A tree, while good for efficiently collecting and distributing data, can create artificial boundaries where none actually exist. Processors that are physically near one another can be topologically distant from one another, depending on their relative positions in the tree, even when the tree is laid out so as to preserve as much of the locality afforded by the communications network as possible. This loss of locality can be alleviated by adding connections in the tree between nodes that are physically near each other. Such a tree, called an X-Tree, is the basic data structure upon which **XTM**'s algorithms are based. Despain, *et. al.* [25, 6], first introduced the X-Tree data structure as a communications network topology. The particular variant of X-Tree we use is a *full-ring X-Tree without end-around connections.*

The rest of this section discusses the need for global information in solving the dynamic thread management problem. We then suggest combining trees as a mechanism for efficiently collecting and disseminating such global information.

3.1.1 The Need for Global Information

As discussed in Chapter 1, thread management is essentially a global optimization problem. Even if we forego optimality, it is easy to see how global knowledge is necessary in order to make good local decisions. For example, in a system where the producers of threads are responsible for deciding where they should be run, the choice of where to send a new thread is strongly influenced by the overall state of the machine. If the machine is relatively "full," then we want to keep a newly created thread near its creator to enhance locality, but if the machine is relatively "empty," we may need to send the thread to some distant processor to improve load-sharing. Similarly, in a system where idle processors are responsible for "drafting" work, the overall state of the machine is just as important. If there is no work to be found on nearby processors, a searcher has to know what regions of the machine contain threads available to be run.

As shown in Chapter 7, diffusion methods and round-robin methods perform relatively poorly on large multiprocessors. Such thread-management algorithms share the attribute that they use no knowledge about the overall state of the machine. This seems to reduce the effectiveness of such methods, since some knowledge of overall machine state is necessary to achieve good load-sharing.

Management of global knowledge can be prohibitively expensive on large-scale machines. The minimum latency for a global broadcast is proportional to the machine diameter. Furthermore, such a broadcast places a load on the communications network at least proportional to the number of processors. If every processor continually produces information that changes the global state, it is clearly unacceptable for each processor to broadcast that information to the all other processors every time a local state change occurs. Similarly, if global information is concentrated on one node, prohibitive hot-spot problems can result. If every processor has to inform that node in the event of a change, and if every processor has to query that node to find out about changes, then the network traffic near that node and the load on the node itself becomes overwhelming, even for relatively small machines.

So the key question is the following: if global information is necessary in order to perform effective thread management, how can we manage that information in such a way as to not put an unacceptable load on any processor or any part of the communications network?

3.1.2 Prospective Solutions

As discussed in Section 1.3, we implement the global Self-Scheduled model [14, 21] by maintaining a queue of runnable threads on every processor. Therefore, the global information useful to a thread-management algorithm could potentially include the state of every one of these queues. However, the cost of keeping every processor informed of the state of every other processor's thread queue would quickly become unmanageable, even for relatively small machines, irrespective of communications architecture. Some way to distill this global information is needed, such that the cost of collecting and disseminating the distilled information is acceptable, while keeping around enough information to make good thread-management decisions.

Software combining [29] presents itself as the obvious way to keep information collection and dissemination costs manageable. When combining techniques are employed, the load on the communications network can be held to an acceptable level. Furthermore, if one is careful, certain combining strategies can guarantee an acceptably low load on all processors, even ones that contain nodes high up in the combining tree. Consequently, we require that the global information used by the thread manager must be organized in such a manner that combining techniques apply: any operation used to distill two separate pieces of global information into one piece must be associative.

The most straightforward way to distill the state of an individual processor queue into one piece of information is to take the length of that queue. A simple sum can then be used to combine two such pieces of data (see Figure 3-1). Each node in such a combining tree keeps track of the sum of the queue lengths for the processors at the leaves of the subtree headed by that node. Unfortunately, it quickly becomes apparent that maintaining an exact sum is both expensive and impossible: impossible because of the communication delays along child-parent links in the tree, and expensive because in such a scheme, nodes near the root of the tree are constantly updating their own state, leaving no time for useful work (execution of threads).

Since maintenance of exact weights in the tree is impossible anyhow, perhaps approximate weights could keep costs acceptably low, while still providing sufficient information for the thread manager to function acceptably. In Chapters 5 and 7, we explore two such ap-

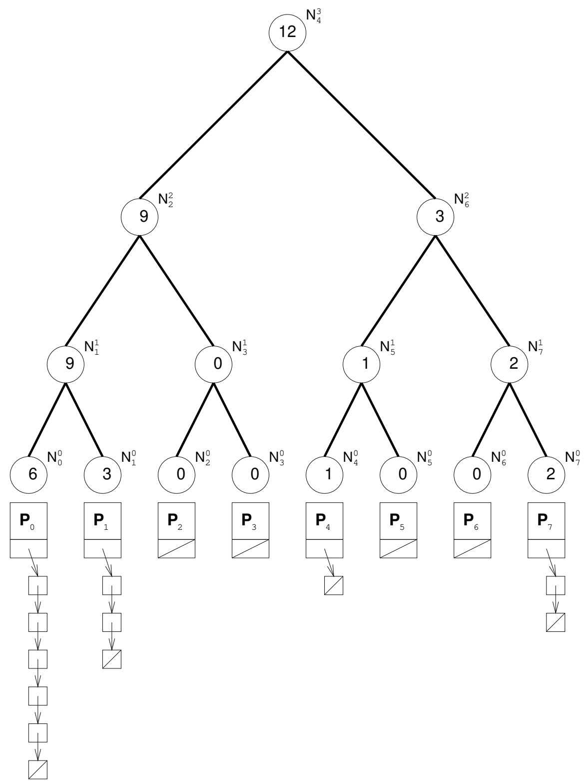


Figure 3-1: Combining Tree with Exact Queue Lengths: *The tree is implemented as a data structure distributed among the various processors in the system. Each node in the tree keeps track of the total number of runnable threads on queues on the processors at the leaves of the subtree headed by that node. The rectangular boxes labeled P_i represent processors in a one-dimensional mesh. Each processor holds a queue of threads; in some cases, that queue is empty. The circles represent nodes in the combining tree. Node N_i^l represents a node at level l in the tree residing on processor P_i. Details of the mapping of tree nodes onto processors are given in Section 4.1.2.*

proximation techniques. The first of these techniques maintains weights at each node of the combining tree as before, but instead of informing its parent whenever its weight changes, a node only informs its parent of a weight change that crosses one of a well-chosen set of boundaries. Details of this technique for disseminating global information are discussed more thoroughly in Section 6.4, under the heading **XTM-C**.

Results in Chapter 7 show that an even simpler approximation technique gives better practical results. Figure 3-2 illustrates this simpler approximation, which reduces the "weight" maintained at each node to a single "presence" bit. A node's presence bit is turned on when any of the processors at the leaves of the subtree headed by that node has at least one runnable thread on its queue.

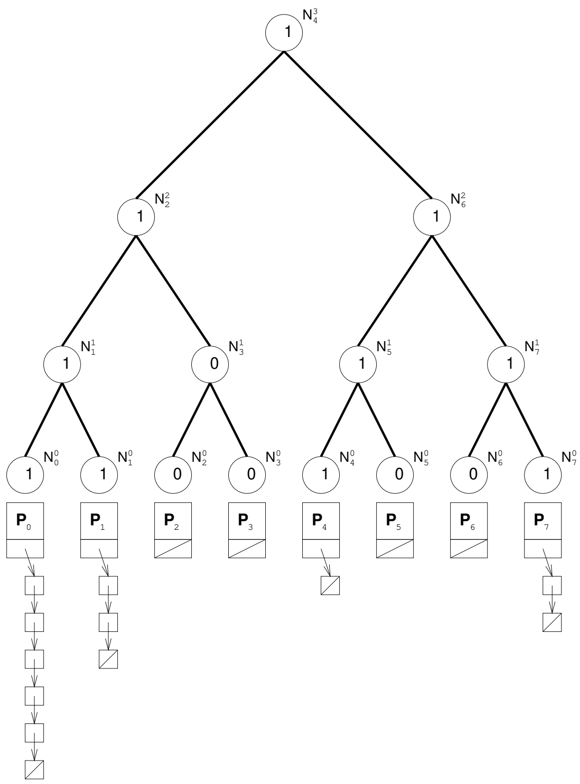


Figure 3-2: Combining Tree with Presence Bits: *The tree is distributed among the processors as described in Figure 3-1. Each node in the tree maintains a "presence bit," which is turned on whenever any queue on any processor at any leaf on the subtree headed by that node contains at least one runnable thread.*

3.2 Matching Threads with Processors

In many dynamic computations, it is unavoidable that situations arise in which one area of the machine is rich with work, with most or all processors busy, while another area is starved for work, with most or all processors idle. In such situations, the communications network can easily be overloaded as work is transferred from the rich area to the sparse area. Furthermore, a tree-based algorithm is prone to hot-spot behavior at higher-level nodes. Both of these problems can be avoided if *combining* is employed: a single request can be made to serve multiple clients.

Combining techniques are not only useful for collecting and disseminating information, they can also be used to collect and disseminate the threads themselves. As an example of why combining is essential, consider the case where one section of a large machine is very busy, containing an excess of runnable threads. At the same time, some other section of the same machine is nearly idle, with very few runnable threads. If combining is not used, then drafting-style thread managers would have each processor in the idle section requesting work from the busy section, while producer-driven managers would have each processor in the busy section sending work over to the idle section. In either case, there would be many messages being sent over long distances.

We propose the use of combining to cut down on the number of long-distance messages. This combining should cause a single message to go from an idle section to a busy section, where all threads to be sent over are gathered, sent back in a single chunk, and then distributed among the idle processors.

3.2.1 X-Tree-Based Design

The need for combining suggests the development of algorithms based on trees: a tree is an ideal structure around which to build algorithms that employ combining. Furthermore, trees are easy to embed in most known architectures in a natural, locality-preserving fashion. However, most communications architectures provided a richer communications structure than that of a tree. Therefore, if simple tree-based designs are used, there is a potential for a loss of locality: the tree can introduce topological boundaries where no physical boundaries exist. Locality lost in this manner can be regained by adding connections in the tree between nodes that are physically near each other. A tree enhanced with such links, introduced in [25] and [6] as a *full-ring X-Tree without end-around connections*, is the basic data structure upon which **XTM**'s algorithms are based (see Figures 4-1 and 4-2).

In Chapter 5, we show that the nearest-neighbor links are needed to get good theoretical behavior. However, results in Chapter 7 demonstrate that for most applications, the simple tree-based design attains significantly higher performance than the X-Tree-based design. For systems in which the ratio of processor speed to network speed is higher, the X-Tree's performance surpasses the simple tree's performance.

3.3 Shared-Memory vs. Message-Passing

We use message-passing as the primary means of interprocessor communication. This reduces **XTM**'s communication requirements, thus increasing performance over a shared-memory implementation. In Chapter 5 these performance gains are shown to become significant on large machines.

One of the earliest design decisions concerned programming style: should programs employ shared-memory or message-passing? This may seem odd, since any algorithm implemented in one of the two styles can be implemented in the other. However, there are compelling theoretical and practical arguments in favor of the message-passing style. This research was performed with the Alewife machine as the primary target machine. Since Alewife supports both efficient shared-memory and message-passing, we had the luxury of choosing between the two styles.

To start with, we give informal definitions of the terms "shared-memory" and "message-passing." Briefly, the central difference between the two mechanisms is whether or not each communication transaction is acknowledged individually. In a message-passing environment, no acknowledgment is required for individual messages; in a sequentially consistent shared-memory environment, every communication transaction requires either a reply containing data or an acknowledgment that informs the requesting processor that the requested transaction is complete.

The two kinds of nodes in a shared-memory environment: processing nodes and memory nodes. Throughout this thesis, when we say shared-memory, we really mean the sequentially consistent shared-memory [17]. Communication takes place in the form of a request from a processor to a memory node, requiring an acknowledgment when the request has been satisfied. The actual low-level particulars of such a transaction depend on such machine-specific details as the existence of caches and the coherence model employed. Requests come in the form of *reads*, which require a reply containing the data being read; *writes*, which modify the data and require an acknowledgment, depending on the machine details, and *read-modify-writes*, which modify the data and require a reply containing the data.

In a message-passing environment, there are only processing nodes. Data is kept in memories local to the processors, and one processor's data is not directly accessible by another processor. Communication takes place in the form of messages between the processors, which require no acknowledgments.

Message-passing systems have the potential to achieve significantly more efficient use of the interprocessor communications network than do shared-memory systems. In many cases, the message overhead required to support the shared-memory abstraction is not needed for correct program behavior. In those cases, a message-passing system can achieve significantly better performance than a shared-memory system [11]. When, however, a large fraction of interprocessor communication is in the *read, write*, or *read-modify-write* form directly supported by a shared-memory system, then using a shared-memory system may yield better performance because shared-memory systems are usually optimized to support such accesses very efficiently. When the distance messages travel in the communications network is taken into account, the advantage held by a message-passing system over a shared-memory system can be significantly greater. For the algorithms employed by **XTM**, the gain can be as high as a factor of $\log p$, where p is the number of processors in the system (see Chapter 5). The advantages of message-passing are not limited to asymptotics. Kranz and Johnson [11] show that the cost of certain primitive operations such as thread enqueueing and dequeueing can improve by factors of five or more when message-passing is used.

If message-passing is so superior to shared-memory, why use shared-memory at all? This question is also addressed in [11]. There are two answers to this question. First, certain types of algorithms are actually *more* efficient when run on shared-memory systems. Second, it seems that the shared-memory paradigm is easier for programmers to handle, in the same way that virtual-memory systems are easier for programmers than overlay systems. Even for the relatively simple algorithms employed by **XTM**, implementation was significantly easier in the shared-memory style. However, the performance gains afforded by message-passing outweighed the additional complexity of implementation, and message-passing was the ultimate choice.

In this chapter, we have presented and argued for a number of early high-level design decisions. In the next chapter, we show how we assembled these decisions to produce a detailed design of a high-performance thread-management system.

Chapter 4

X-Tree-Based Thread Management

This thesis presents a thread distribution system based on an *X-Tree-driven search.* An X-Tree [25, 6] is a tree augmented with links between same-level nodes. The particular variant we use contains *near-neighbor* links between *touching* same-level nodes. In the parlance used in [25] and [6], this is a full-ring X-Tree without end-around connections (see Figures 4-1 and 4-2). In this chapter, we describe in detail the algorithms that go into **XTM**, an **X**-Tree-based **T**hread **M**anager.

When a new thread is created, its existence is made public by means of *presence bits* in the X-Tree. When a node's presence bit is set, that means that there are one or more runnable threads somewhere in the sub-tree rooted at that node; a cleared presence bit indicates a sub-tree with no available work. Presence information is added to the X-Tree as follows: when a thread is either created or enabled (unblocked), it is added to the *runnable thread queue* associated with some processor. If the queue was previously empty, the presence bit in the leaf node associated with that processor is set. The presence bits in each of the node's ancestors are then set in a recursive fashion. The process continues up the X-Tree until it reaches a node whose presence bit is already set. In Chapter 5, we show that although this algorithm can cost as much as $\mathcal{O}(nk)$,[1] the expected cost is

[1] n is the mesh dimensionality; k is the mesh radix.

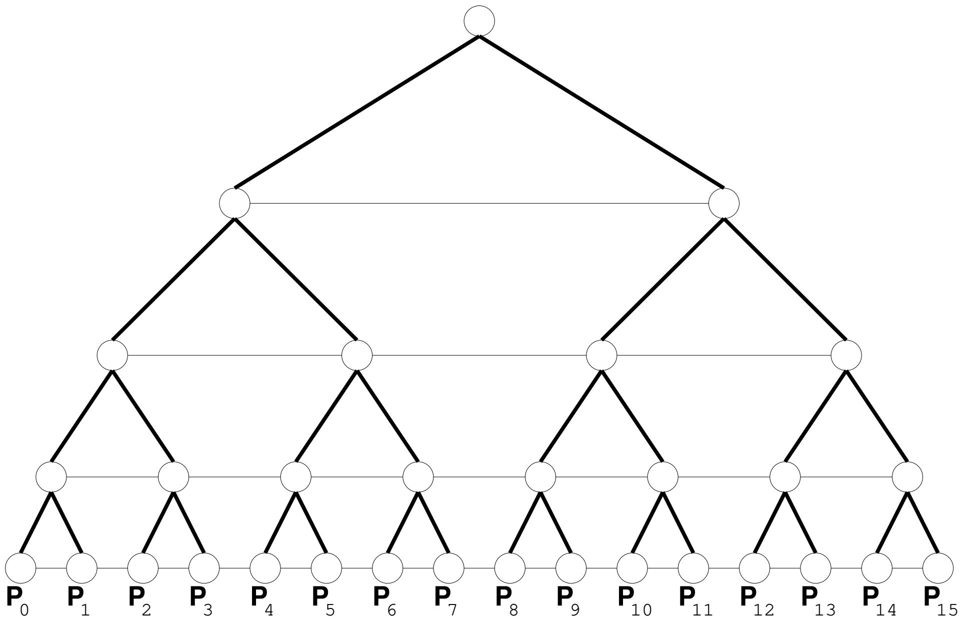


Figure 4-1: Binary X-Tree on a One-Dimensional Mesh Network: *Labels of the form* $\mathbf{P}_i$ *indicate physical processors. Circles indicate nodes in the X-Tree. Thick lines indicate parent-child links. Thin lines indicate near-neighbor links.*

actually $\mathcal{O}\left(\frac{n}{\lambda^2}\right)$, when the distributions of thread queue lengths are independent, identically distributed random variables with probabilities of non-zero length of at least λ, for some $\lambda : \text{Prob}\left(\text{qlen} > 0\right) \geq \lambda$. Whenever a thread is taken off of a queue, if that causes queue to become empty, a similar update process is performed, with similar costs.

When a processor becomes idle, it initiates a *thread search*. A thread search recursively climbs the X-Tree, examining successively larger neighborhoods in the process. When a given node is examined, it and all of its neighbors are queried as to the status of their presence bits. If none of the bits is set, the search continues with the node's parent. Otherwise, for one of the nodes whose presence bits are set, the searcher requests half of the work available on the sub-tree rooted at that node. Such a request is satisfied by recursively requesting work from each child whose presence bit is set. In Chapter 5, we show that this search algorithm is guaranteed to finish in $\mathcal{O}\left(nd\right)$ time, where n is the dimensionality of the mesh and d is the distance between the searcher and the nearest non-empty queue. This is $\mathcal{O}\left(n\right)$-competitive with an optimal drafting-style thread manager. Furthermore, in order to limit long-distance accesses, the search algorithm combines requests from multiple

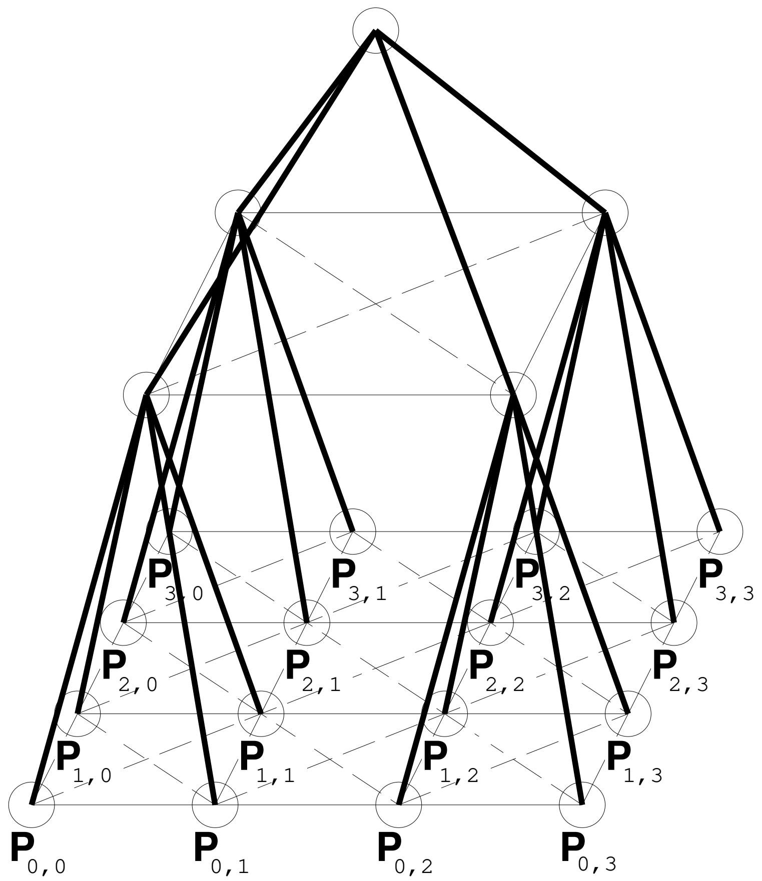


Figure 4-2: Quad-X-Tree on a Two-Dimensional Mesh Network: *Labels of the form* $\mathbf{P}_{i_0,i_1}$ *indicate physical processors. Circles indicate nodes in the X-Tree. Thick lines indicate parent-child links. Thin solid lines indicate near-neighbor links to edge-adjacent neighbors. Thin dashed lines indicate near-neighbor links to corner-adjacent neighbors.*

children of each node: for a given sub-tree, only a single representative searches outside that sub-tree at a time. When the representative comes back with a collection of runnable threads, the threads are divided among the other searchers that were blocked awaiting the representative's return.

XTM is based on an X-Tree data structure embedded into the communications network. The nodes of the X-Tree contain presence bits whose values are updated whenever a thread is either created or consumed. The presence bits are used to drive a search process that gets runnable threads to idle processors. Section 4.1 describes the details of the X-Tree, including its embedding in the network. Section 4.2 gives the details of the presence bit update algorithm. Section 4.3 describes the process that sends runnable threads to idle processors.

Finally, Section 4.4 gives an example of these algorithms in action. All algorithms described in this chapter are assumed to employ the message-passing communication style.

4.1 X-Trees

One- and two-dimensional X-Trees are pictured in Figures 4-1 and 4-2. For the algorithm presented in this thesis, an X-Tree is a software data structure. The nodes that make up the X-Tree are distributed around the communications network on the processors. The X-Tree nodes include one leaf node on each processor and enough higher-level nodes to make up the rest of the tree. The higher-level nodes are distributed throughout the machine in such a way as to keep nodes that are topologically near to one another in the tree physically near to one another in the network (see Section 4.1.2). The X-Tree guides the process of matching idle processors with runnable threads, using near-neighbor links between nodes that are physically near to each other. On an n-dimensional mesh network, each node in the X-Tree has up to $3^n - 1$ near-neighbor links. Each leaf of the X-Tree is associated with a physical processor, and is stored in the memory local to that processor.

4.1.1 Notation

The X-Tree data structure is embedded in the communications network. The individual nodes that make up the X-Tree are resident on the processors. Each node is labeled with its *level* in the tree and the mesh coordinates of the processor it resides on. Tree levels start at zero at the leaves, one at the next higher level and so on. A node is identified as:

$$N^l_{i_0,i_1,\ldots}$$

where l is the level of the node in the tree and $i_0, i_1, \ldots$ are the mesh coordinates of the processor on which the node resides ($P_{i_0,i_1,\ldots}$). This notation does not distinguish between two or more same-level nodes on a single processor. This is not a problem because we are not interested in any embeddings that put two or more same-level nodes on the same processor: any embedding that puts more than one same-level node on a single processor will tend not to distribute tree management costs as well as embeddings that put only one node at each level on a single processor; experience shows that systems that distribute their overheads poorly tend to perform poorly overall.

The Manhattan distance through the mesh between two nodes $N^{l}_{i_0,i_1,\ldots}$ and $N^{m}_{j_0,j_1,\ldots}$ is written:

$$\mathcal{D}\left(N^{l}_{i_0,i_1,\ldots}, N^{m}_{j_0,j_1,\ldots}\right) = |j_0 - i_0| + |j_1 - i_1| + \ldots$$

We assume that the cost of communicating between two nodes is equal to the Manhattan distance between the nodes. This is true, for example, for the e-cube routing scheme [26].

We use different notation to refer to the distance covered when taking the shortest path through the X-Tree between two nodes:

$$\mathcal{X}\left(N^{l}_{i_0,i_1,\ldots}, N^{m}_{j_0,j_1,\ldots}\right)$$

In other words, while $\mathcal{D}\left(N^{l}_{i_0,i_1,\ldots}, N^{m}_{j_0,j_1,\ldots}\right)$ is the shortest distance between $N^{l}_{i_0,i_1,\ldots}$ and $N^{m}_{j_0,j_1,\ldots}$ through the mesh, while $\mathcal{X}\left(N^{l}_{i_0,i_1,\ldots}, N^{m}_{j_0,j_1,\ldots}\right)$ is the shortest distance between the two nodes when traversing the X-Tree.

The m^{th} ancestor of node $N^{0}_{i_0,i_1,\ldots}$ is written:

$$A^{m}_{i_0,i_1,\ldots}$$

More generally, the m^{th} ancestor of any node $N^{l}_{i_0,i_1,\ldots}$, which is the $m+l^{th}$ of node $N^{0}_{i_0,i_1,\ldots}$, is written:

$$A^{l+m}_{i_0,i_1,\ldots}$$

We use this notation to discuss relationships between X-Tree nodes and their ancestors. In particular, we need to discuss costs associated with climbing the tree from a node to one of its ancestors.

The relationship between a node ($N^l_{i_0,i_1,...}$) and its parent ($A^{l+1}_{i_0,i_1,...} = N^{l+1}_{i'_0,i'_1,...}$) can be expressed through a set of (usually simple) embedding functions:

$$i'_0 = f_0\left(i_0, i_1, ..., l\right);\;\; i'_1 = f_1\left(i_0, i_1, ..., l\right);\;\; ...$$

Clearly, the costs associated with traversing an X-Tree depend on how the tree is embedded into the communications network. These embedding functions are used to formalize a particular embedding scheme. Formal statements of embedding functions re used in formal algorithm statements and in mathematical proofs.

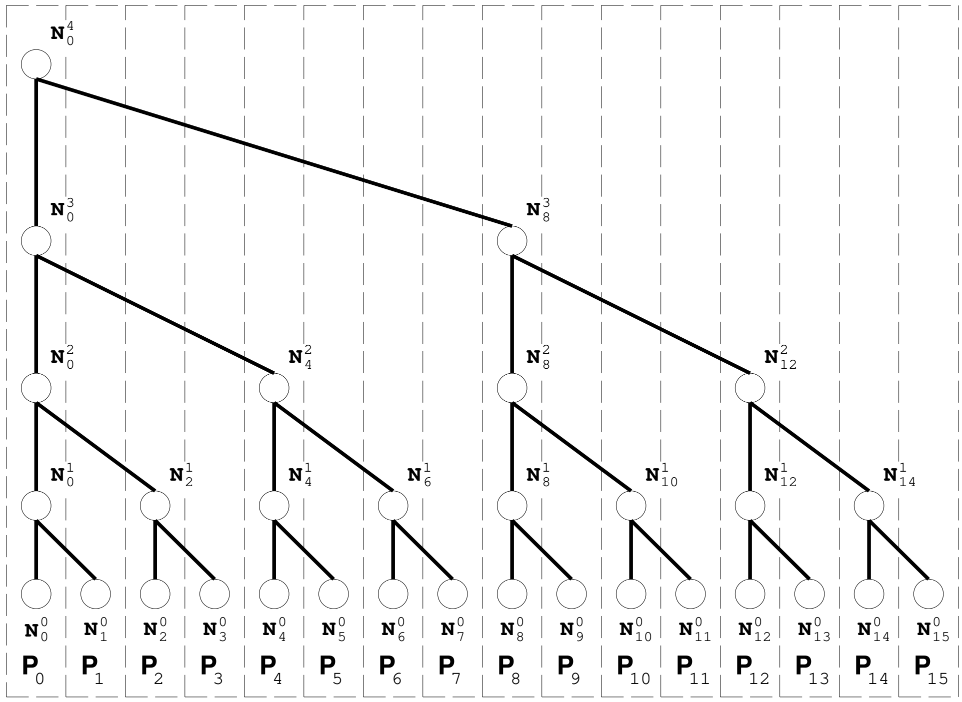


Figure 4-3: Embedding a Binary Tree in a One-Dimensional Mesh: *Naive Solution*.

4.1.2 Embedding a Tree in a k-ary n-Dimensional Mesh

An X-Tree embedding has to capture the locality in the communications network while balancing the overhead its management costs place on the individual processors. The locality captured by the X-Tree depends on its embedding in the underlying space: a "good" embedding places nodes that are topologically near one another in the X-Tree physically near one another in the network. The X-Tree should also be distributed so as to minimize the maximum load placed on any individual processor by any part of the thread-management algorithm.

One-Dimensional Case

Figure 4-3 illustrates a straightforward embedding of a binary tree in a one-dimensional mesh. Each processor's leaf node is resident on that processor. The following expression captures the relationship between a node and its ancestors:

$$A_i^l = N_{i'}^l; \;\; i' = i \wedge \left(-2^l\right)$$

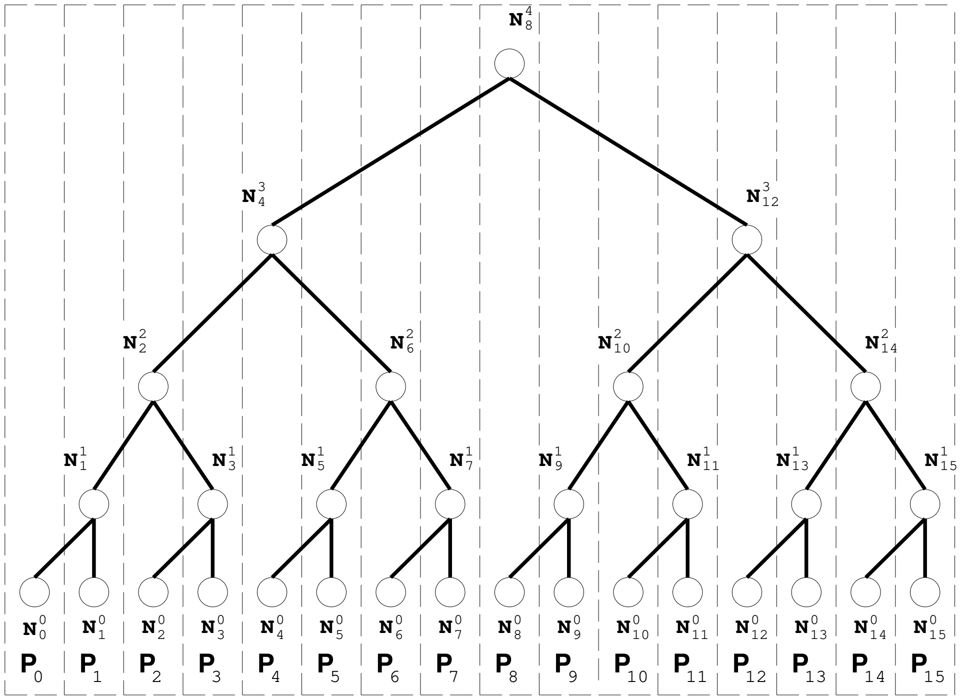


Figure 4-4: Embedding a Binary Tree in a One-Dimensional Mesh: *Better Solution*.

Using this embedding, if a node on $\mathbf{P}_i$ has an ancestor at level l, that ancestor is on $\mathbf{P}_{i'}$, where i' is simply i with l low-order bits masked off. This embedding has two major drawbacks. First, worst-case tree-traversal costs between leaf nodes are worse than they need to be. Second, note that some processors (*e.g.*, $\mathbf{P}_0$) contain more nodes of the tree than others. This leads to higher network contention near those heavily used nodes, cutting down on overall performance. Also, the processors and memories on those nodes are more heavily loaded down with thread management overhead, leading to even more unbalance and worse performance.

Figure 4-4 illustrates a better embedding of a binary tree in a one-dimensional mesh:

$$A_i^l = N_{i'}^l; \;\; i' = \left[i \wedge \left(-2^{l-1}\right)\right] \vee 2^{l-1} \quad (\text{for } l \geq 1)$$

Using this embedding, if a node on $\mathbf{P}_i$ has an ancestor at level l, that ancestor is on $\mathbf{P}_{i'}$, where i' is i with $l-1$ low-order bits masked off and with the l^{th} bit set to one. This gives a better worst-case tree traversal cost than the first embedding. Furthermore, it guarantees that at most two tree nodes are resident on any given processor, yielding better hot-spot behavior.

The distance between any two nodes in a one-dimensional mesh is:

$$\mathcal{D}\left(N_i^l, N_j^m\right) = |j - i| \,.$$

For the suggested embedding, the distance between any node at level m and its parent is:

$$\mathcal{D}\left(N_i^m, A_i^{m+1}\right) = 2^{m-1},$$

except at the leaves of the tree.

The distance between a leaf node and its parent is:

$$\mathcal{D}\left(N_i^0, A_i^1\right) = \text{either } 0 \text{ or } 1,$$

depending on whether the parent is on the same processor as the child or not.

In the one-dimensional case, all of a node's near neighbors are the same distance from the node:

$$\mathcal{D}\left(N_i^m, N_{i\pm 2^m}^m\right) = 2^m.$$

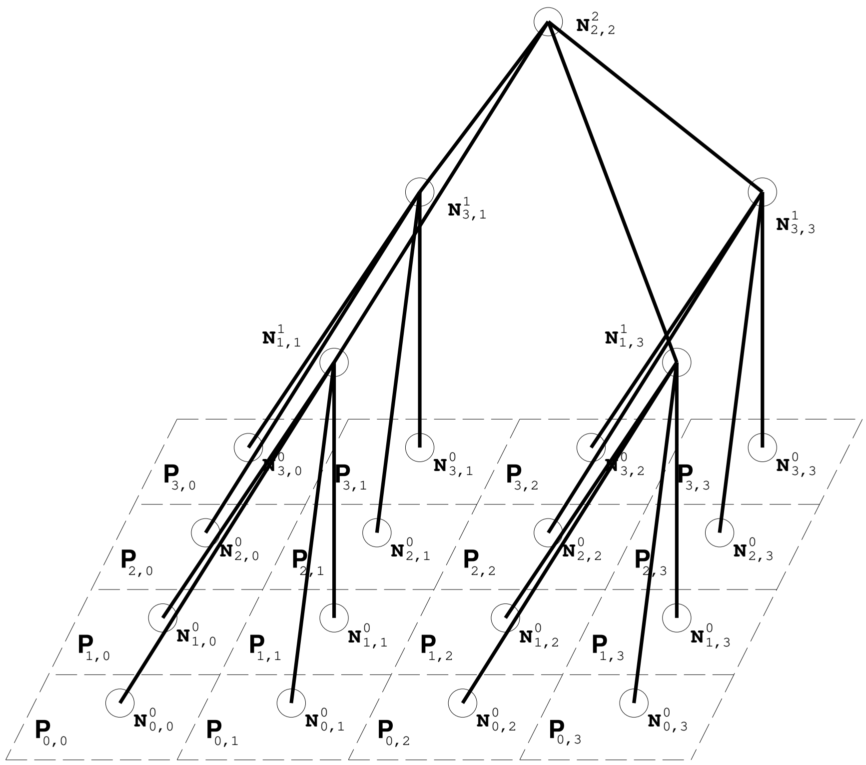


Figure 4-5: Embedding a Quad-Tree in a Two-Dimensional Mesh.

Two-Dimensional Case

Figure 4-5 illustrates a well-distributed embedding of a quad-tree in a two-dimensional mesh:

$$A^l_{i_0,i_1} = N^l_{i'_0,i'_1}; \;\; i'_0 = \left[i_0 \wedge \left(-2^{l-1}\right)\right] \vee 2^{l-1}; \;\; i'_1 = \left[i_1 \wedge \left(-2^{l-1}\right)\right] \vee 2^{l-1} \quad (\text{for } l \geq 1)$$

Using this embedding, if a node on $\mathbf{P}_{i,j}$ has an ancestor at level l, that ancestor is on $\mathbf{P}_{i',j'}$. i' is i with $l-1$ low-order bits masked off and with the l^{th} bit set to one; j' is j with $l-1$ low-order bits masked off and with the l^{th} bit set to one.

For any two-dimensional embedding, the distance between any two nodes is:

$$\mathcal{D}\left(N^l_{i_0,i_1}, N^m_{j_0,j_1}\right) = |j_0 - i_0| + |j_1 - i_1|\,.$$

For the suggested embedding, the distance between any node at level m and its parent is:

$$\mathcal{D}\left(N^m_{i_0,i_1}, A^{m+1}_{i_0,i_1}\right) = 2\left(2^{m-1}\right),$$

except at the leaves of the tree.

The distance between a leaf node and its parent is:

$$\mathcal{D}\left(N^0_{i_0,i_1}, A^1_{i_0,i_1}\right) = \text{either } 0,\ 1 \text{ or } 2,$$

depending whether the parent is on the same processor as the child or not.

In the two-dimensional case, a node has two classes of near neighbors: those that are *edge-adjacent* and those that are *vertex-adjacent*. Each node has up to four edge-adjacent neighbors and up to four vertex-adjacent neighbors. For example, in Figure 4-2, node $N^0_{0,1}$ has five near neighbors: $N^0_{0,0}$, $N^0_{1,0}$, $N^0_{1,1}$, $N^0_{1,2}$ and $N^0_{0,2}$. $N^0_{0,0}$, $N^0_{1,1}$ and $N^0_{0,2}$ are edge-adjacent; $N^0_{1,0}$ and $N^0_{1,2}$ are vertex-adjacent.

The distance between any node at level m and its face-adjacent nearest neighbors is:

$$\mathcal{D}\left(N^m_{i_0,i_1}, N^m_{i_0\pm 2^m,i_1}\right) = \mathcal{D}\left(N^m_{i_0,i_1}, N^m_{i_0,i_1\pm 2^m}\right) = 2^m.$$

The distance between any node at level m and its vertex-adjacent nearest neighbors is:

$$\mathcal{D}\left(N^m_{i_0,i_1}, N^m_{i_0\pm 2^m,i_1\pm 2^m}\right) = 2\left(2^m\right).$$

n-dimensional Case

It would be difficult to illustrate an tree embedded in an n-dimensional mesh. However, we can generalize the results presented in earlier sections to obtain the following embedding functions for $l \geq 1$:

$$\begin{aligned}
A^l_{i_0,i_1,\ldots,i_{n-1}} &= N^l_{i'_0,i'_1,\ldots,i'_{n-1}} \\
i'_0 &= \left[i_0 \wedge \left(-2^{l-1}\right)\right] \vee 2^{l-1} \\
i'_1 &= \left[i_1 \wedge \left(-2^{l-1}\right)\right] \vee 2^{l-1} \\
i'_{n-1} &= \left[i_{n-1} \wedge \left(-2^{l-1}\right)\right] \vee 2^{l-1}
\end{aligned}$$

Using this embedding, if a node on $\mathbf{P}_{i_0,i_1,\ldots i_{n-1}}$ has an ancestor at level l, that ancestor is on $\mathbf{P}_{i'_0,i'_1,\ldots i'_{n-1}}$. All the i'_x's are obtained by taking the corresponding i_x, masking off the $l-1$ low-order bits and setting the l^{th} bit to one.

For any n-dimensional embedding, the distance between any two nodes is:

$$\mathcal{D}\left(N^l_{i_0,i_1,\ldots,i_{n-1}}, N^m_{j_0,j_1,\ldots,j_{n-1}}\right) = |j_0 - i_0| + |j_1 - i_1| + \ldots + |j_{n-1} - i_{n-1}|\,.$$

For the given embedding, the distance between any node at level m and its parent is:

$$\mathcal{D}\left(N^m_{i_0,i_1,\ldots,i_{n-1}}, A^{m+1}_{i_0,i_1,\ldots,i_{n-1}}\right) = n\left(2^{m-1}\right),$$

except at the leaves of the tree.

The distance between a leaf node and its parent is:

$$\mathcal{D}\left(N^0_{i_0,i_1,\ldots,i_{n-1}}, A^1_{i_0,i_1,\ldots,i_{n-1}}\right) = \text{either } 0,\ 1,\ \ldots,\ \text{or } n.$$

In the n-dimensional case, a node has n classes of near neighbors, each of which is a different distance from the node. We label these classes with the index i, where i ranges from 1 to n. The maximum number of neighbors in class i is:

$$\begin{pmatrix} n \\ i \end{pmatrix} 2^i.$$

Note that this yields a maximum total of $3^n - 1$ neighbors.

The distance between any node at level m and a neighbor of class i is:

$$i\left(2^m\right).$$

4.2 Updating Information in the X-Tree

Every processor has its own runnable thread queue. When a new thread is created, it is added to the creating processor's queue. Similarly, when a thread is enabled (unblocked), it is added to the queue of the processor on which it most recently ran. Threads can be moved from one processor's queue to another by a thread search initiated by an idle processor (see below). Before a thread is run, it is removed from its queue.

When a thread is added to or taken from a queue, this fact is made public by means of *presence bits* in the X-Tree. When a node's presence bit is set, that means that there are one or more runnable threads to be found somewhere in the sub-tree rooted at that node; a cleared presence bit indicates a sub-tree with no available work. Of course, since the X-Tree is a distributed data structure, updating this presence information can not occur instantaneously; as long as an update process is in progress, the presence information in the tree is not absolutely accurate. Therefore, the thread distribution algorithm can only use these presence bits as *hints* to guide a search; it must not depend on their absolute accuracy at all times.

Presence information is disseminated as follows: when a runnable thread queue goes either from *empty* to *non-empty* or from *non-empty* to *empty*, an update process is initiated. This process recursively climbs the tree, setting or clearing presence bits on its way up. A presence-bit-setting update process terminates when it reaches a node whose presence bit is set; likewise, a presence-bit-clearing update process terminates when it reaches a node that has some other child whose presence bit is set. Appendix A gives a more formal pseudocode description of the presence bit update algorithm.

Update processes have the responsibility of making it globally known that work is available. These efforts are combined at each node of the tree. The world only needs to be informed when a queue goes either from *empty* to *non-empty* or from *non-empty* to *empty*. When that happens, presence bits in the tree are set or cleared by an update process that recursively climbs the tree, terminating when it reaches a node whose presence bit state is consistent with its children's states. When more than one update process arrives at a given node in the tree at one time, the processes are executed atomically with respect to each other. Assuming that no new information is received between the execution of these processes, only the first process ever proceeds on up the tree. In this way, the information distribution algorithm combines its efforts at the nodes of the tree.

Whenever a presence bit in a node changes state, the node's neighbors and parent are all informed. The presence-bit status of each of a node's neighbors and children is cached locally at the node, decreasing search time significantly. Of course, this increases the cost of distributing information throughout the tree, but as Chapter 5 shows, the expected cost of information distribution is very low due to the decreasing likelihood of having to update nodes higher up in the tree.

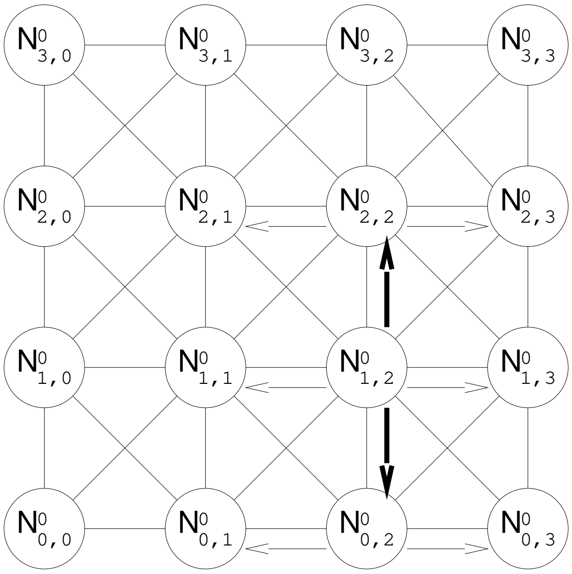


Figure 4-6: Two-Dimensional Presence Bit Cache Update: *When the state of a node's presence bit changes, it informs its nearest neighbors, which cache presence bit information in order to cut down on usage of the communications network. This information is disseminated using a simple n-level multicast tree algorithm. Thick arrows indicate the first phase of the information dissemination process ($N^0_{1,2}$ informs $N^0_{0,2}$ and $N^0_{2,2}$); thin arrows indicate the second phase ($N^0_{1,2}$ informs $N^0_{1,3}$ and $N^0_{1,1}$, while $N^0_{0,2}$ informs $N^0_{0,3}$ and $N^0_{0,1}$, and $N^0_{2,2}$ informs $N^0_{2,3}$ and $N^0_{2,1}$). In an n-dimensional system, there are n such phases.*

Cached presence bits are updated in a divide-and-conquer fashion. The updating node first informs its two nearest near-neighbors in the first dimension. Then all three nodes inform their nearest near-neighbors in the second dimension. This continues until the n^{th} dimension is complete. Figure 4-6 illustrates the cached presence bit update algorithm.

4.3 Thread Search

An idle processor initiates a thread search process, which traverses the tree looking for runnable threads to execute. It starts at the leaf node local to the idle processor.

When examining a given node, a searcher first checks the state of the node's presence bit. If it is set, then there is work somewhere in the sub-tree rooted at the node. If not, the presence bits of the nodes' neighbors are examined. If none of them are set, the searcher starts again one level higher in the tree. If more than one searcher arrives at the same node, then only one *representative* continues the search, and the rest wait for the representative to find work for them all.

When a searcher encounters a node whose presence bit is set, the searcher goes into gathering mode. A gatherer collects work from all of the leaves under the node where the gatherer originated. It starts by requesting work from those of the node's children whose presence bits are set. The children request work from their children, and so on down to the leaves. Each leaf node sends half of its runnable threads back to its parent, which combines the work thus obtained from all of its children, sending it on up to its parent, and so on, back up to the node where the gathering process originated.

Finally, the set of threads thus obtained is distributed among the waiting searchers. As a representative searcher makes its way back down the tree to the processor that spawned it, at each level, it hands an equal share of the work off to other searchers awaiting its return.

4.4 An Example

Figures 4-7 through 4-15 illustrate the thread search process on an 8-ary 1-dimensional mesh. In all of these figures, X-Tree nodes are represented by circles labeled N_i^l. Numbers inside the circles indicate the state of the presence bits associated with each node. X-Tree nodes are joined by thin lines representing near-neighbor links and thick lines indicating parent-child links.

At each leaf node in the X-Tree, there is a square processor box labeled $\mathbf{P}_i$ indicating that the leaf node is associated with processor i. A shaded square indicates that the processor has useful work to do; a non-shaded square indicates an idle processor. Under each processor square is a small rectangle representing the runnable thread queue associated with the processor. A queue rectangle with a line through it is empty; a non-empty queue points to a list of squares indicating runnable threads.

Throughout this example, we disregard the fact that presence bits are cached by parents and neighbors. This has no effect on the functioning of the search algorithm; it is simply an optimization that improves search costs.

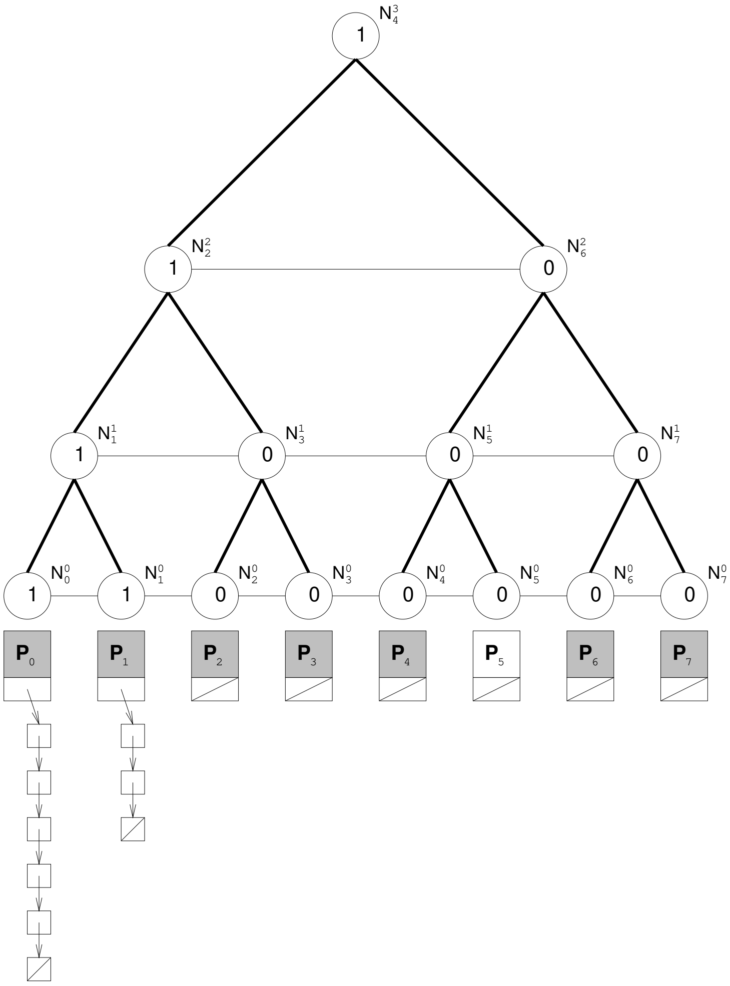


Figure 4-7: Thread Search Example – I.

We start with processors $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$, $\mathbf{P}_4$, $\mathbf{P}_6$ and $\mathbf{P}_7$ busy and processor $\mathbf{P}_5$ idle. All runnable thread queues are empty except for those on $\mathbf{P}_0$ and $\mathbf{P}_1$, which contain six and three threads, respectively.

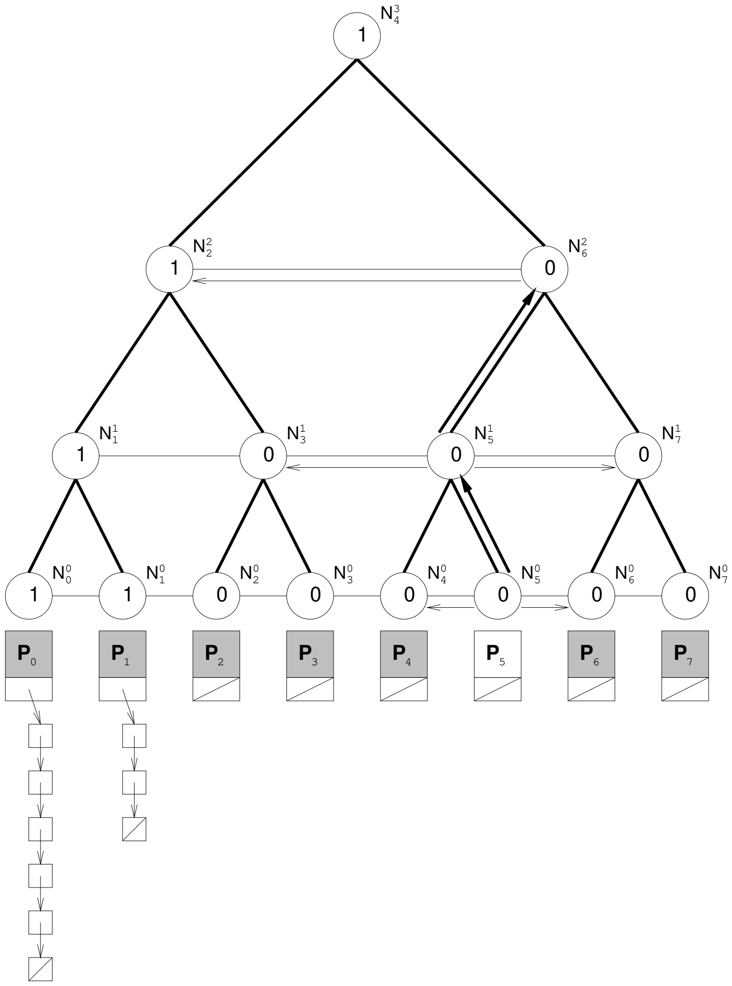


Figure 4-8: Thread Search Example – II.

$\mathbf{P}_5$, which is idle, initiates a thread search. The search process first examines $\mathbf{P}_5$'s leaf node (N_5^0), where it finds no work. It then examines N_5^0's two nearest neighbors, N_4^0 and N_6^0, finding both of their presence bits to be zero, so it continues one level higher in the tree with node N_5^1. N_5^1's presence bit is zero, as are those of both its neighbors, N_3^1 and N_7^1, so the search continues one level higher at node N_6^2. N_6^2's presence bit is zero, but its neighbor, N_2^2 has its presence bit set, so the search goes into gathering mode.

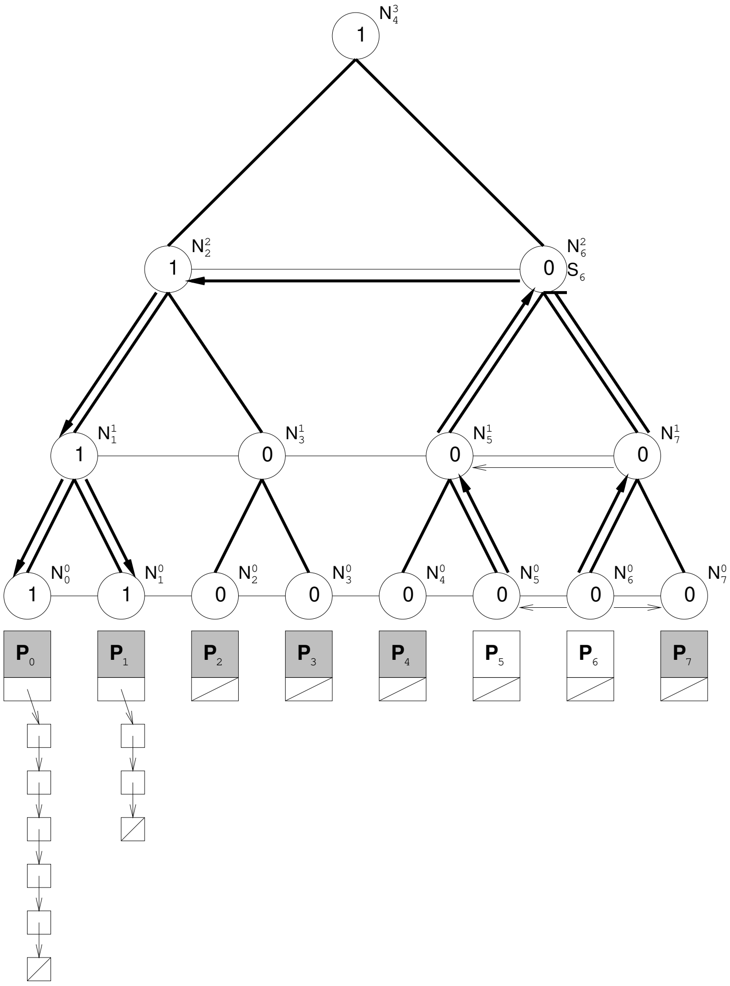


Figure 4-9: Thread Search Example – III.

The search process that originated on $\mathbf{P}_5$ initiates a gathering process starting at node N_2^2. A request for work is sent to node N_1^1, the one child of N_2^2 whose presence bit is set. Requests for work are then sent to nodes N_0^0 and N_1^0, both children of N_1^1 and both of which have set presence bits.

Meanwhile, $\mathbf{P}_6$ has become idle, presumably because the thread it was executing either blocked or terminated. It initiates a second thread search process, which examines the presence bits of the following nodes: N_6^0, N_5^0 and N_7^0, N_7^1 and N_5^1, finding them all to be zero. When it gets to node N_6^2, it finds that the other search process is searching outside the immediate neighborhood, so it waits for the other searcher to return with work. The waiting searcher is indicated in the figure by the symbol S_6.

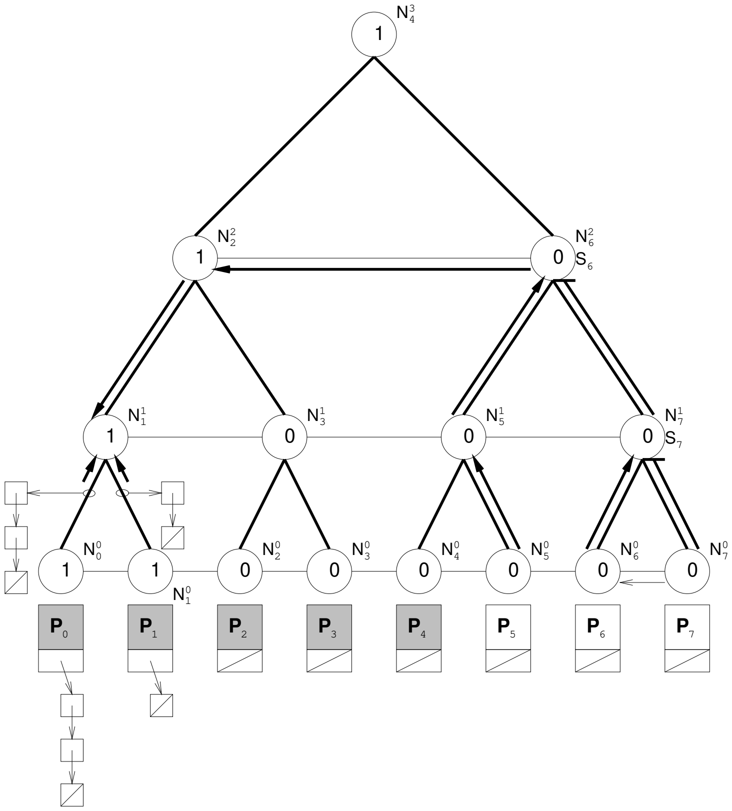


Figure 4-10: Thread Search Example – IV.

The gathering process initiated at node N_2^2 has reached the leaves of the tree and now goes into gather mode. Requests for work that went to nodes N_0^0 and N_1^0 cause half of the work on $\mathbf{P}_0$'s and $\mathbf{P}_1$'s queues to be detached and sent back up the tree towards the requester.

Meanwhile, yet another processor ($\mathbf{P}_7$) has become idle. It initiates a third thread search process, which examines the presence bits of nodes N_7^0 and N_6^0, which are both still zero. When it gets to node N_7^1, it waits for the second searcher to return with work. The waiting searcher is indicated in the figure by the symbol S_7.

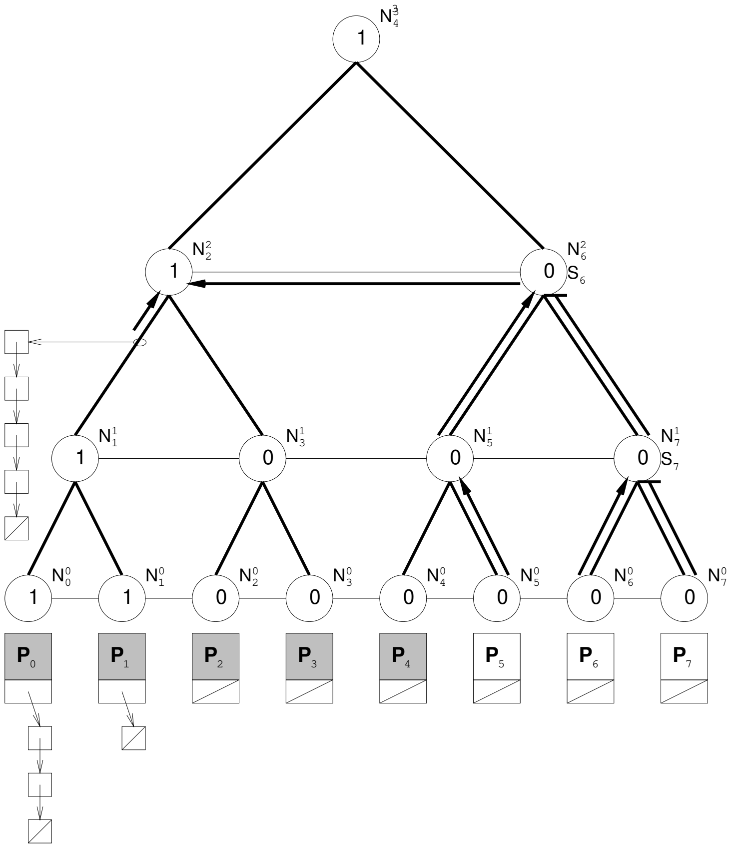


Figure 4-11: Thread Search Example – V.

The gather process continues. The threads taken from processors $\mathbf{P}_0$ and $\mathbf{P}_1$ are combined at node N_1^1 and sent on up to node N_2^2, where the gathering process started.

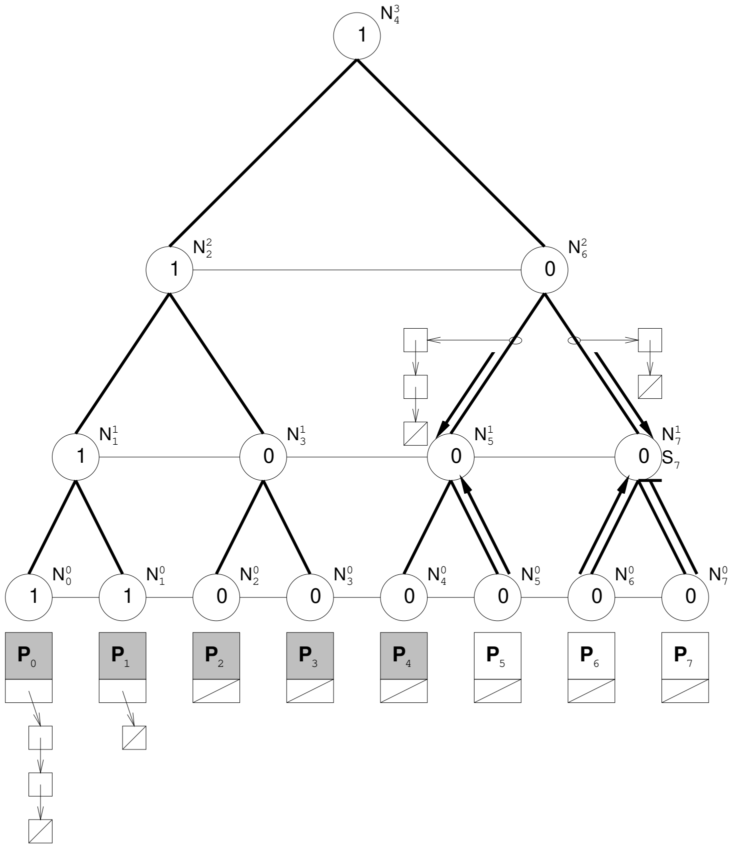


Figure 4-12: Thread Search Example – VI.

The threads are sent back to node N_6^2, where they are split between searcher S_5, which initiated the gathering process, and searcher S_6, which waited back at node N_6^2 for S_5's return.

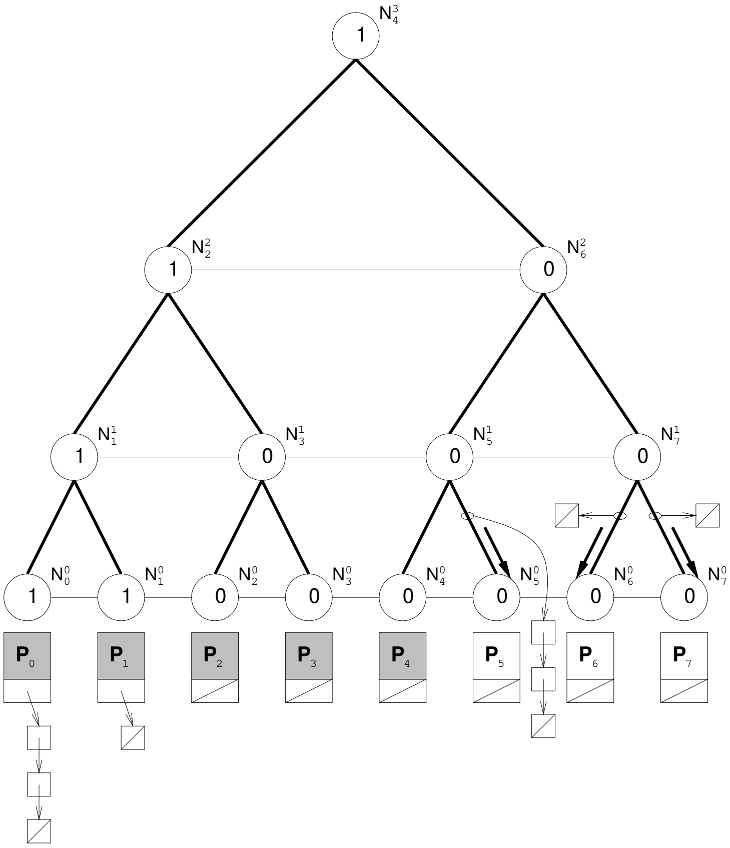


Figure 4-13: Thread Search Example – VII.

Searcher S_5 sends its threads one link closer to $\mathbf{P}_5$. Meanwhile, the work brought back by searcher S_6 is split up between S_6 and S_7, which was waiting at node N_7^1 for S_6's return.

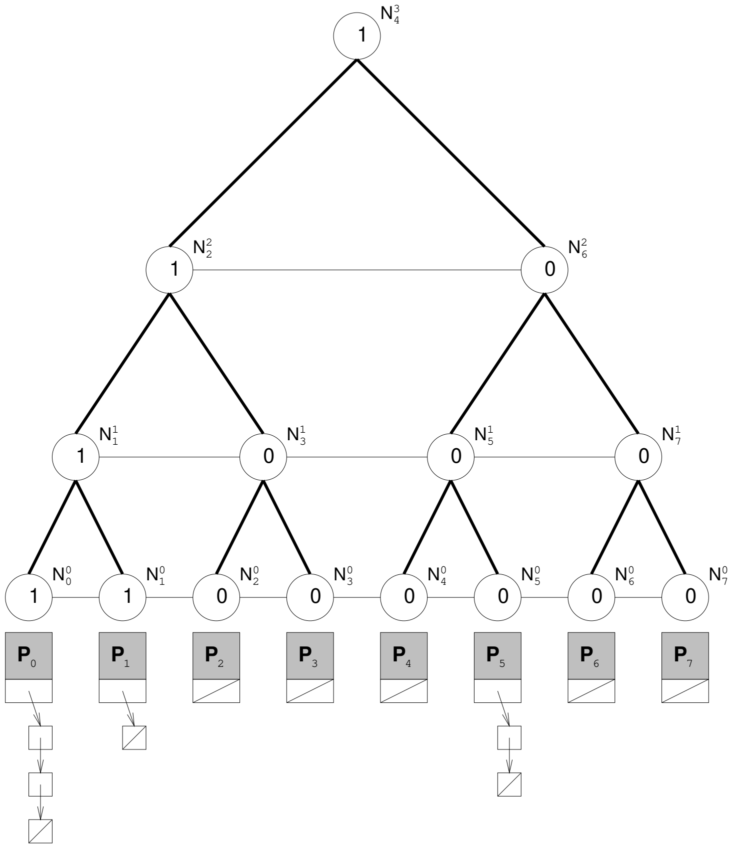


Figure 4-14: Thread Search Example – VIII.

All searchers have returned to their originating processors. At each processor, the first thread brought back is run and the rest are put on the runnable thread queue. Note that the presence bits in the tree have not yet been updated to reflect the new work that has just become available.

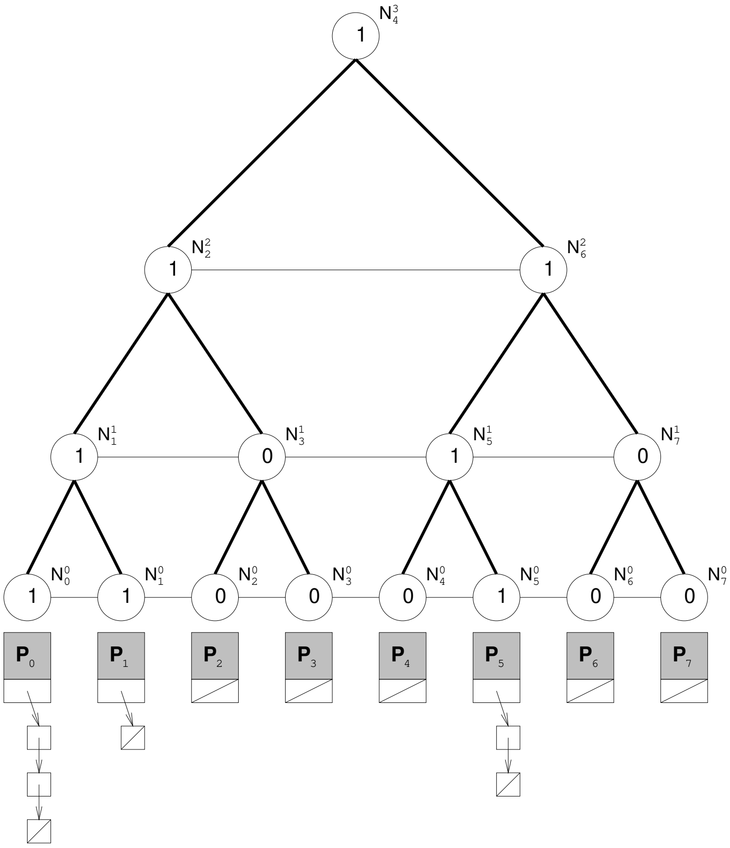


Figure 4-15: Thread Search Example – IX.

A tree update process is sent out from $\mathbf{P}_5$ because its runnable thread queue, which was previously empty, now contains threads that may be run by other processors. The process updates nodes N_5^0, N_5^1 and N_6^2. N_4^3's presence bit was already set and therefore did not have to be modified.

This example illustrated the thread search algorithm in action. Note that a single search process brought back work for three searching processors from the same neighborhood.

In this chapter, we gave a detailed description of the two algorithms at the heart of **XTM**: the global information update algorithm and the thread search algorithm. In the next chapter, we give asymptotic analyses for the two algorithms.

Chapter 5

Analysis

In this chapter, we analyze the various pieces of **XTM** in terms of both execution time and network bandwidth consumed. In particular, we look at the thread search and presence-bit-update algorithms, analyzing the one-dimensional case, the two-dimensional case and the n-dimensional case for each.

Our most important results are the following:

1. On a machine with a sufficiently high, balanced workload, the expected cost of maintaining presence bits in the X-Tree is proved to be asymptotically constant, regardless of machine size.

2. The algorithm that matches runnable threads with idle processors is shown to be eight-competitive with an idealized drafting-style adversary, running on a two-dimensional mesh network.

3. The message-passing communication style is shown to yield fundamental improvements in efficiency over a shared-memory style. For the matching process, the advantage is a factor of $\log l$, where l is the distance between an idle processor and the nearest runnable thread.

In addition, we give asymptotic cost bounds for **XTM**'s search and update algorithms on one-, two- and n-dimensional mesh networks. We give results in terms of maximum latency and bandwidth requirements.

Unless otherwise stated, we assume that an efficient message-passing mechanism is employed, *i.e.*, a message can be sent from any processor in the machine to any other with no acknowledgment necessary [15]. When such a message represents an active entity moving about the system, then decisions about where it is to be sent next can be made without having to communicate with the processor from which the message first originated.

In addition, all analyses of search costs in this chapter assume that information in the tree is accurate. This is in fact an approximation: due to delays in the communications network, in a dynamic environment, it is impossible to guarantee that dynamic data structures are consistent at all times. The logic of the presence bit update algorithm guarantees that soon after the global state stops changing, the information in the tree will be accurate and consistent. It would be interesting to speculate as to the effect of the temporary inaccuracies of the information in the tree.

Finally, all analyses assume that local computation is free; all costs are measured in terms of communication latency. To obtain execution time, we look at the (serialized) communication requirements of the algorithm's critical path, assuming that non-critical-path pieces of the algorithm do not interfere with the critical path. We also assume that all messages are the same size, therefore consuming the same network bandwidth per unit distance. This is again an approximation: in fact, the more work that is moved, the more network bandwidth is consumed. However, for the Alewife machine, most of the bandwidth consumed by a message is start-up cost: it takes far longer to set up a path than it does to send flits down a path that has already been set up. For this reason, we stay with the constant-message-size approximation.

5.1 One-Dimensional Case

The one-dimensional case is easiest to illustrate. This section presents in-depth analyses of the various pieces of the thread-distribution algorithm as implemented on a one-dimensional mesh. In Section 5.2, we extend the analyses to two dimensions; in Section 5.3, we perform the same analyses for an n-dimensional mesh.

Throughout this section, the minimum distance d between two nodes on processors $\mathbf{P}_i$ and $\mathbf{P}_j$ is simply $|j - i|$:

$$d = \mathcal{D}\left(N_i^0, N_j^0\right) = |j - i|$$

This is the lowest possible cost for communicating between $\mathbf{P}_i$ and $\mathbf{P}_j$.

Also, when following the shortest path through the tree between nodes N_i^0 and N_j^0, we have to ascend l levels. l, which is an integer, is either $\lfloor \log_2 d \rfloor$ or $\lceil \log_2 d \rceil$, depending on how the two nodes are aligned with respect to the tree.

$$\lfloor \log_2 d \rfloor \leq l \leq \lceil \log_2 d \rceil$$

Finally, L is the height of the tree:

$$\lfloor log_2 k \rfloor \leq L \leq \lceil log_2 k \rceil$$

5.1.1 Search

The search algorithm can be executed using either the message-passing style or the shared-memory style. In this section, we derive lower bounds on the latency using both styles. We show that for the message-passing style, the X-Tree-guided search algorithm is four-competitive with the optimal. For the same algorithm, the shared-memory style yields results that are worse by a factor of $\log d$, where d is the distance between a processor and the nearest available work.

Message-Passing

Here, we show that the cost of searching the X-Tree for nearby work is four-competitive with the optimal adversary. In a k-ary one-dimensional mesh, a path from a node N_i^0 to node N_j^0 is established by ascending l levels in the tree from node N_i^0 to its level-l ancestor A_i^l, crossing over to A_j^l and descending back down to node N_j^0. The following analysis demonstrates the competitive factor of four.

The distance covered when taking shortest path through the tree between nodes is the following:

$$\begin{aligned}
\mathcal{X}\left(N_i^0, N_j^0\right) &= \mathcal{D}\left(N_i^0, A_i^l\right) + \mathcal{D}\left(A_i^l, A_j^l\right) + \mathcal{D}\left(A_j^l, N_j^0\right) \\
&\leq \left(1 + \sum_{x=1}^{l-1} 2^{x-1}\right) + 2^l + \left(1 + \sum_{x=1}^{l-1} 2^{x-1}\right) \\
&\leq 2^{l-1} + 2^l + 2^{l-1} \\
&\leq 2\left(2^l\right) \\
&\leq 2\left(2^{\lceil \log_2 d \rceil}\right) \\
&\leq 4d
\end{aligned}$$

Therefore, the traversal cost for the X-Tree is no more than a factor of four worse than direct access through the mesh.

So far, we have given an upper bound on the tree traversal cost between two leaf nodes. The actual search process is a bit more complicated for two reasons. First, once a runnable thread is found, it is has to travel back to the requesting processor. Second, since the X-Tree is used for combining as well as search-guiding purposes, it employs a *gathering* process once work is located, so that *half* of the work on the subtree with work is sent back to the requesting sub-tree, to be distributed among all requesting processors from that sub-tree. The gathering process first broadcasts requests to the entire sub-tree rooted at the node and then waits for replies, which are combined at the intermediate nodes between the source of the gathering process and the leaves of the sub-tree rooted at that node. Since the gathering process executes in a divide-and-conquer fashion, the latency for the gathering process is the same as the time it takes for a single message to be sent to a leaf from the originating node, and for a reply to be sent back. Therefore, the critical path for the search algorithm costs at most $8d$, which is twice as much as the worst-case traversal calculated above.

The optimal adversary simply sends a request to the nearest processor containing work.

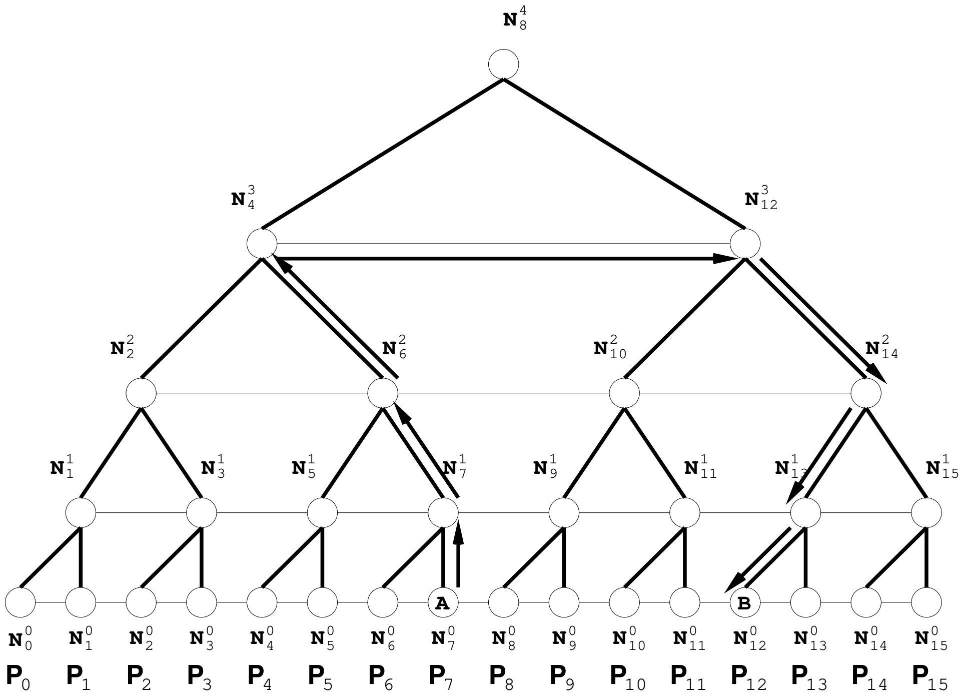


Figure 5-1: One-Dimensional Search – Worst Case: *A direct path from node* **A** *(N_7^0) to node* **B** *(N_{12}^0) covers five network hops. The corresponding tree traversal covers 15 network hops: $N_7^0 \rightarrow N_7^1 \rightarrow N_6^2 \rightarrow N_4^3 \rightarrow N_{12}^3 \rightarrow N_{14}^2 \rightarrow N_{13}^1 \rightarrow N_{12}^0$.*

The destination processor then sends some of its work back to the requester. If the requesting processor is $\mathbf{P}_i$ and the destination processor is $\mathbf{P}_j$, then this entire process has a cost of $2d$. This gives us a competitive factor of four when comparing the X-Tree-based search to the optimal adversary.

Figure 5-1 gives an example of a worst-case tree traversal scenario on a 16-processor one-dimensional mesh: the tree traversal is three times more expensive than the direct path. Similar examples on larger meshes approach the worst-case factor of four.

We now derive the worst-case network bandwidth requirement for a search that begins at N_i^0 when the closest work is at N_j^0. The bandwidth consumed is broken into four pieces: ascending the tree to the lowest level that has a neighbor that has work, sending a "gather" message to that neighbor, gathering half the available work from that neighbor, and sending that work back to the original source of the request.

$$
\begin{aligned}
\mathcal{B}\left(N_i^0, N_j^0\right) &\leq \left(1 + \sum_{x=1}^{l-1} 2^{x-1}\right) + 2^l + \left(1 + \sum_{x=0}^{l-1} 2^{l-x} 2^{x-1}\right) + 2^l + \left(1 + \sum_{x=1}^{l-1} 2^{x-1}\right) \\
&\leq 2^{l-1} + 2^l + l2^{l-1} + 2^l + 2^{l-1} \\
&\leq (l+6)\left(2^{l-1}\right) \\
&\leq (7 + \log_2 d)\left(2^{\lceil \log_2 d \rceil - 1}\right) \\
&\leq (7 + \log_2 d)\, d
\end{aligned}
$$

The network bandwidth consumed by the search process is $\mathcal{O}(d \log d)$, as compared to a running time of $\mathcal{O}(d)$. This is the expected behavior, due to the use of a gathering process to collect threads from the entire sub-tree rooted at A_j^l.

Shared-Memory

When shared-memory processing is used, the matching algorithm becomes more expensive. The path from a node N_i^0 to node N_j^0 is established by ascending l levels in the tree from node N_i^0 to its level-l ancestor A_i^l, crossing over to A_j^l and descending back down to node N_j^0. Each step in the algorithm requires communication between the original source processor $\mathbf{P}_i$ and a different node in the X-Tree.

$$\begin{aligned}
\mathcal{X}\left(N_i^0, N_j^0\right) &= \sum_{x=1}^{l} \mathcal{D}\left(N_i^0, A_i^x\right) + \sum_{x=l}^{0} \mathcal{D}\left(N_i^0, A_j^x\right) \\
&\geq -1 + \left(\sum_{x=0}^{l-1} 2^x\right) + \left(\sum_{x=l}^{0} 2^{l-1}\right) \\
&\geq (l+1)\, 2^{l-1} - 1 \\
&\geq \left(\lfloor \log_2 d \rfloor + 1\right) 2^{\lfloor \log_2 d \rfloor - 1} - 1 \\
&\geq (\log_2 d)\, \frac{d}{4} - 1
\end{aligned}$$

In other words, for the shared-memory case, the cost is $\Omega\left(d \log d\right)$. This is more expensive than the idealized adversary by a factor of $(\log d)$. This cost derives from repeated long-distance accesses through the communications network as the search closes in on its quarry.

5.1.2 Presence Bit Update

In this section, we derive lower bounds on the latency using both shared-memory and message-passing communication styles. We show that for the message-passing style, the X-Tree-guided update algorithm has a worst-case cost proportional to the network diameter, but an expected cost independent of machine size, given a machine load that is both balanced and sufficiently high. The asymptotic behavior of the update algorithm is not nearly as sensitive to the communication style as the search algorithm. We show that the worst case cost only goes up by a factor of two when the shared-memory communication style is used.

Message-Passing

Whenever a runnable thread queue changes state between *empty* and *non-empty*, that information is recorded in the form of presence bits in the tree. Three sub-tasks are performed at each level in the tree:

1. Determine whether the presence bit at this node needs to be modified. If so, set this node's presence bit to the new value and continue with step two. If not, exit.

2. Direct this node's neighbors to change their cached presence bit copies for this node to the new value.

3. Inform this node's parent – continue the update process by starting this algorithm at step one on the parent node.

Step 1 incurs no cost: it only requires local calculations. Step 2 is not included in the critical path analysis because it is not in the algorithm's critical path and can execute concurrently. Therefore, it is only the child-parent communication cost that affects the algorithm's critical-path cost.

The update cost can be characterized in terms of worst case and expected behavior. We show that although the worst-case cost is proportional to the network diameter, the expected cost is independent of machine size for certain machine load conditions. First, the bad news: $\mathcal{U}$, the *worst-case* tree update critical-path cost, can be as bad as $\mathcal{O}(k)$. We now show why this is the case.

$$
\begin{aligned}
\mathcal{U} &\leq 1 + \sum_{i=0}^{L-2} 2^i \\
&\leq 2^{L-1} \\
&\leq 2^{\lceil \log_2 k \rceil - 1} \\
&\leq k
\end{aligned}
$$

Note that the worst-case tree update cost is proportional to the network diameter. We will find that this is also true in the general (n-dimensional) case.

Now, the good news: the *expected* tree update cost is much lower. We calculate the expected cost for a *sufficiently loaded, balanced* machine. By balanced, we mean that every thread queue has the same probability distribution for its queue length. By sufficiently loaded, we mean that there exists some non-zero constant lower bound on the probability that the queue is non-empty: λ. In other words, *every* processor's thread queue has a distribution of queue lengths such that the probability of non-zero queue length is at least λ, for some $\lambda : \text{Prob}\,(\text{qlen} > 0) \geq \lambda$. In the following derivation, $\mathrm{E}[\mathcal{U}]$ signifies the expected update cost, $\text{Prob}\,(\text{lev} = i)$ represents the probability that the algorithm makes it to exactly the i^{th} level and $\mathcal{C}\,(i)$ is the expected cost of communicating between a node at the i^{th} level and its parent. Finally, let $\mu = (1 - \lambda)$.

$$
\begin{aligned}
\mathrm{E}[\mathcal{U}] &= \sum_{i=0}^{L-2} \mathcal{C}\,(i)\,\text{Prob}\,(\text{lev} = i) \\
&\leq 2^{-1}\left[\mu^{(2^0)} - \mu^{(2^1)}\right] + 2^0\left[\mu^{(2^1)} - \mu^{(2^2)}\right] \\
&\quad +\ldots + 2^{L-4}\left[\mu^{(2^{L-3})} - \mu^{(2^{L-2})}\right] + 2^{L-3}\left[\mu^{(2^{L-2})}\right] \\
&\leq \mu 2^{-2} + \sum_{i=0}^{L-2} 2^{i-2}\mu^{(2^i)} \\
&\leq \frac{1}{4}\left[\mu + \sum_{i=0}^{L-2} 2^i \mu^{(2^i)}\right] \\
&\leq \frac{1}{4}\left[\mu + \sum_{j=1}^{\infty} j\mu^j\right] \\
&\leq \frac{\mu}{4}\left[1 + \frac{d}{d\mu}\sum_{j=0}^{\infty} \mu^j\right] \\
&\leq \frac{\mu}{4}\left[1 + \frac{d}{d\mu}\frac{1}{1-\mu}\right] \\
&\leq \frac{\mu}{4}\left[1 + \frac{1}{(1-\mu)^2}\right] \\
&\leq \frac{(1-\lambda)}{4}\left[1 + \frac{1}{\lambda^2}\right]
\end{aligned}
$$

Most of this derivation is straightforward algebraic manipulation. The only subtlety concerns the transition from $\sum_{i=0}^{L-2} 2^i \mu^{(2^i)}$ to $\sum_{j=1}^{\infty} j\mu^j$. The second of the two expressions is simply a restatement of the the first expression, adding some (strictly positive) terms to the sum and changing the summation index.

This result shows that although the worst-case tree update cost is proportional to the network diameter, on a machine whose workload is both balanced and sufficiently high, the expected update cost is $\mathcal{O}\left(\frac{1}{\lambda^2}\right)$, which *does not* depend on the network diameter.

The network bandwidth requirement for the tree update process is somewhat higher; not only does it include child-parent communication costs, but it also includes costs incurred when updating neighbors' cached presence bits.

$$\begin{aligned} \mathcal{B}(\mathcal{U}) &\leq k + \sum_{i=0}^{L-1} 2\left(2^i\right) \\ &\leq k + 2\left(2^{\lceil \log_2 k \rceil}\right) \\ &\leq k + 4k \end{aligned}$$

Like the critical-path cost, the worst-case bandwidth requirement for the tree update process is $\mathcal{O}(k)$, differing from the critical-path cost by a constant factor only. We will find that this constant factor is a function of n, the network dimensionality.

The expected bandwidth requirement for the tree update process is also significantly lower than the worst case:

$$\begin{aligned} \mathrm{E}[\mathcal{B}(\mathcal{U})] &= \mathrm{E}[\mathcal{U}] + \mathrm{E}[\mathcal{B}(\text{neighbor-cache update})] \\ &\leq \frac{(1-\lambda)}{4}\left[1 + \frac{1}{\lambda^2}\right] + \sum_{i=0}^{L-1} 2\left(2^i\right)\mu^{\left(2^i\right)} \\ &\leq \frac{(1-\lambda)}{4}\left[1 + \frac{1}{\lambda^2}\right] + 2\mu \frac{d}{d\mu}\frac{1}{1-\mu} \\ &\leq \frac{(1-\lambda)}{4}\left[1 + \frac{1}{\lambda^2}\right] + \frac{2(1-\lambda)}{\lambda^2} \end{aligned}$$

On a machine whose workload is both balanced and sufficiently high, the expected bandwidth requirement is $\mathcal{O}\left(\frac{1}{\lambda^2}\right)$, which is independent of k, the network diameter. Note that the expected update bandwidth differs from the critical-path bandwidth only by a constant factor. In this case as well, we will see that this factor is a function of n.

Shared-Memory

For the matching process, the cost became fundamentally higher when going to the shared-memory programming style. For the tree update process, this is *not* the case; the rise in cost is a constant factor.

$$\begin{aligned}
\mathcal{U} &\leq 1 + \sum_{i=1}^{L-1} 2^i \\
&\leq 2^L \\
&\leq 2^{\lceil \log_2 k \rceil} \\
&\leq 2k
\end{aligned}$$

In the case of message-passing, the worst-case tree update cost was $\mathcal{O}(k)$. Here in the shared-memory case, the worst-case update cost is also $\mathcal{O}(k)$.

5.2 Two-Dimensional Case

This section presents in-depth analyses of the various pieces of the thread-distribution algorithm as implemented on a two-dimensional mesh. Two-dimensional mesh networks provide more communication capacity than one-dimensional networks. Also, both one- and two-dimensional networks can scale to arbitrarily large sizes without encountering any theoretical snags related to wire lengths, wire densities or heat dissipation. Networks of three or more dimensions have problems existing in real space, due to wire packing, wire lengths and heat dissipation issues.

Throughout this section, the minimum distance d between two nodes on processors $\mathbf{P}_{i_0,i_1}$ and $\mathbf{P}_{j_0,j_1}$ is simply $|j_0 - i_0| + |j_1 - i_1|$:

$$d = \mathcal{D}\left(N^0_{i_0,i_1}, N^0_{j_0,j_1}\right) = |j_0 - i_0| + |j_1 - i_1|$$

This is the lowest possible cost for communicating between $\mathbf{P}_{i_0,i_1}$ and $\mathbf{P}_{j_0,j_1}$.

Also, when following the shortest path through the tree between nodes $N^0_{i_0,i_1}$ and $N^0_{j_0,j_1}$, we have to ascend l levels:

$$\max\left(\lfloor \log_2 |j_0 - i_0| \rfloor, \lfloor \log_2 |j_1 - i_1| \rfloor\right) \leq l \leq \max\left(\lceil \log_2 |j_0 - i_0| \rceil, \lceil \log_2 |j_1 - i_1| \rceil\right)$$

$$\lfloor \max\left(\log_2 |j_0 - i_0|, \log_2 |j_1 - i_1|\right) \rfloor \leq l \leq \lceil \max\left(\log_2 |j_0 - i_0|, \log_2 |j_1 - i_1|\right) \rceil$$

$$\lfloor \log_4 d \rfloor \leq l \leq \lceil \log_2 d \rceil$$

The exact value for l depends on how the nodes are aligned with respect to the tree.

Finally, L is the height of the tree:

$$\lfloor log_2 k \rfloor \leq L \leq \lceil log_2 k \rceil$$

5.2.1 Search

In this section, we derive lower bounds on the latency and bandwidth costs using the message-passing style. We also show that for the message-passing style, the X-Tree-guided search algorithm is eight-competitive with the optimal. For the same algorithm, the shared-memory style yields results that are worse by a factor of $\log d$, where d is the distance between a processor and the nearest available work. Proof of this fact follows the derivation for the one-dimensional network.

Message-Passing

In this section, we show that the cost of searching the X-Tree for nearby work is eight-competitive with the optimal adversary. In a k-ary two-dimensional mesh, a path from a node $N^0_{i_0,i_1}$ to node $N^0_{j_0,j_1}$ is established by ascending l levels in the tree from node $N^0_{i_0,i_1}$ to node $A^l_{i_0,i_1}$, crossing over to node $A^l_{j_0,j_1}$ and descending back down to node $N^0_{j_0,j_1}$. The following analysis demonstrates the competitive factor of eight:

$$\begin{aligned}
\mathcal{X}\left(N^0_{i_0,i_1}, N^0_{j_0,j_1}\right) &= \mathcal{D}\left(N^0_{i_0,i_1}, A^l_{i_0,i_1}\right) + \mathcal{D}\left(A^l_{i_0,i_1}, A^l_{j_0,j_1}\right) + \mathcal{D}\left(A^l_{j_0,j_1}, N^0_{j_0,j_1}\right) \\
&\leq \left[1 + \sum_{x=1}^{l-1} 2\left(2^{x-1}\right)\right] + 2\left(2^l\right) + \left[1 + \sum_{x=1}^{l-1} 2\left(2^{x-1}\right)\right] \\
&\leq 2\left(2^{l-1} + 2^l + 2^{l-1}\right) \\
&\leq 4\left(2^l\right) \\
&\leq 4\left(2^{\lceil \log_2 d \rceil}\right) \\
&\leq 8d
\end{aligned}$$

Therefore, the traversal cost for the X-Tree is no more than a factor of eight worse than direct access through the mesh.

As is shown for the one-dimensional case, both for the X-Tree algorithm and for the optimal adversary, a search costs twice as much as a simple message send from source to destination. Therefore, the competitive factor of eight holds for a search as well as for a one-way message send.

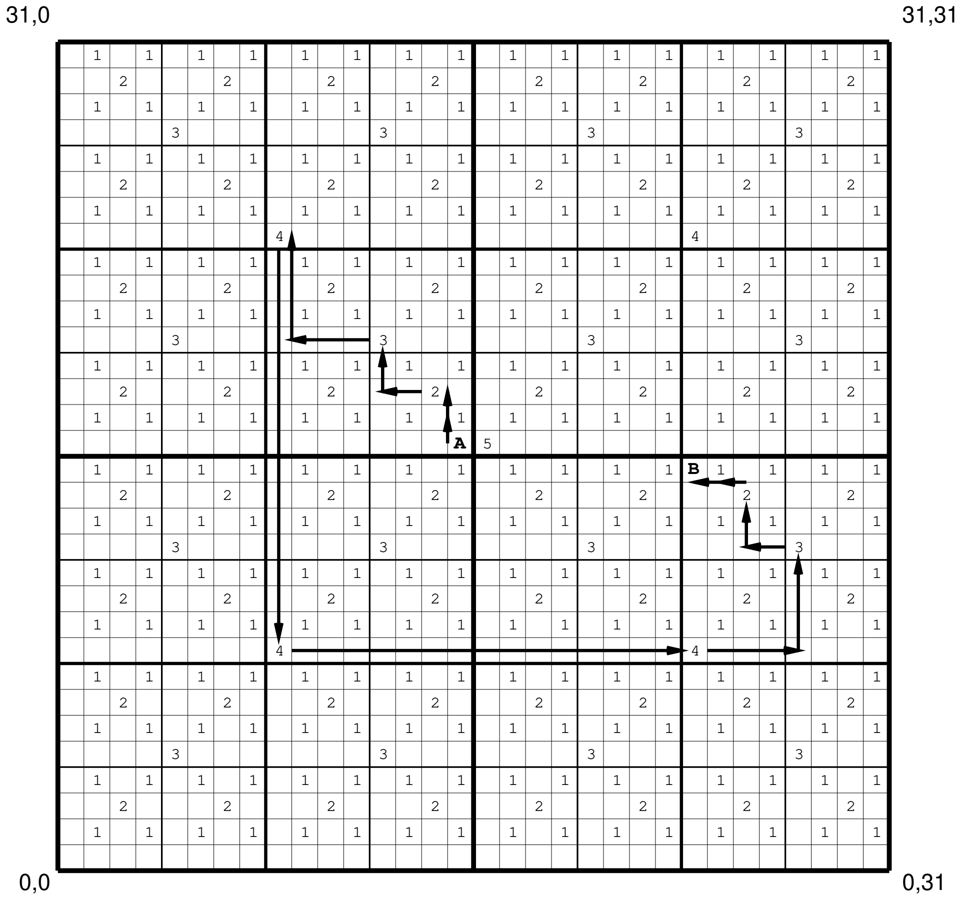


Figure 5-2: Two-Dimensional Search – Worst Case: *A direct path from node* **A** *($N^0_{16,15}$) to node* **B** *($N^0_{15,24}$) covers ten network hops. The corresponding tree traversal covers 62 network hops:* $N^0_{16,15} \rightarrow N^1_{17,15} \rightarrow N^2_{18,14} \rightarrow N^3_{20,12} \rightarrow N^4_{24,8} \rightarrow N^4_{8,24} \rightarrow N^3_{12,28} \rightarrow N^2_{14,26} \rightarrow N^1_{15,25} \rightarrow N^0_{15,24}$.

Figure 5-2 gives an example of a worst-case tree traversal scenario on a 32 by 32 mesh: the tree traversal is more than six times more expensive than the direct path. Similar examples on larger meshes approach the worst-case factor of eight.

We now derive the worst-case network bandwidth requirement for a search that begins at $N^0_{i_0,i_1}$ when the closest work is at $N^0_{j_0,j_1}$:

$$\begin{aligned}
\mathcal{B}\left(N^0_{i_0,i_1}, N^0_{j_0,j_1}\right) &\leq \left(1+\sum_{x=1}^{l-1} 2\left(2^{x-1}\right)\right) + 2\left(2^l\right) + \left[1+\sum_{x=0}^{l-1}\left(4^{l-x}\right)(2)\left(2^{x-1}\right)\right] \\
&\leq 2\left(2^{l-1}+2^l+l4^{l-1}\right) \\
&\leq 2\,(l+3)\left(4^{l-1}\right) \\
&\leq 2\left[3+\lceil log_2 d\rceil\right]\left[4^{\lceil log_2 d\rceil -1}\right] \\
&\leq 2\left[4+\log_2 d\right] d^2
\end{aligned}$$

Although the latency for the search is $\mathcal{O}(d)$, where d is the distance between the searcher and the nearest work, the network bandwidth consumed is $\mathcal{O}\left(d^2 \log d\right)$. This is not surprising, due to the use of a gathering process to collect threads from the entire sub-tree rooted at $A^l_{j_0,j_1}$.

Shared-Memory

The shared-memory analysis for the two-dimensional case follows the message-passing analysis in the same way that it does for the one-dimensional case. The resulting cost goes up by a factor of $\log d$ for the matching algorithm.

5.2.2 Presence Bit Update

In this section, we derive lower bounds on the latency and bandwidth using the message-passing communication style. We the X-Tree-guided update algorithm has a worst-case cost proportional to the network diameter, but an expected cost independent of machine size, for certain load conditions. We argue that the worst case cost only goes up by a factor of two when the shared-memory communication style is used. The derivation of this fact follows that given for the one-dimensional case.

Message-Passing

The presence bit update analysis for two dimensions is similar to that for one dimension. First, the bad news: the *worst-case* tree update critical-path cost can be as bad as $\mathcal{O}(k)$. We now show why this is the case:

$$\begin{aligned} \mathcal{U} &\leq 2\left(1 + \sum_{i=0}^{L-2} 2^i\right) \\ &\leq 2\left(2^{L-1}\right) \\ &\leq 2\left(2^{\lceil \log_2 k \rceil - 1}\right) \\ &\leq 2k \end{aligned}$$

Again, we find that the worst-case tree update cost is proportional to the network diameter.

Now, the good news: again, the *expected* tree update cost is much lower on a machine whose workload is both balanced and sufficiently high. These load conditions are expressed by *any* distribution of thread queue lengths such that the probability of non-zero queue length is at least λ, for some $\lambda : \text{Prob}\,(\text{qlen} > 0) \geq \lambda$.

$$
\begin{aligned}
\mathrm{E}[\mathcal{U}] &= \sum_{i=0}^{L-2} \mathcal{C}\,(i_0, i_1)\, P\,(i_0, i_1) \\
&\leq 2\Big(\, 2^{-1}\left[\mu^{\left(4^0\right)} - \mu^{\left(4^1\right)}\right] + 2^0\left[\mu^{\left(4^1\right)} - \mu^{\left(4^2\right)}\right] \\
&\qquad +\ldots + 2^{L-4}\left[\mu^{\left(4^{L-3}\right)} - \mu^{\left(4^{L-2}\right)}\right] + 2^{L-3}\left[\mu^{\left(4^{L-2}\right)}\right]\Big) \\
&\leq 2\left(\mu 2^{-2} + \sum_{i=0}^{L-2} 2^{i-2}\mu^{\left(4^i\right)}\right) \\
&\leq \frac{2}{4}\left[\mu + \sum_{i=0}^{L-2} 2^i \mu^{\left(4^i\right)}\right] \\
&\leq \frac{1}{2}\left[\mu + \sum_{i=0}^{L-1} 4^i \mu^{\left(4^i\right)}\right] \\
&\leq \frac{1}{2}\left[\mu + \sum_{j=1}^{\infty} j\mu^j\right] \\
&\leq \frac{\mu}{2}\left[1 + \frac{1}{(1-\mu)^2}\right] \\
&\leq \frac{(1-\lambda)}{2}\left[1 + \frac{1}{\lambda^2}\right]
\end{aligned}
$$

This result shows again that although the worst-case tree update cost is proportional to the network diameter, on a machine whose workload is both balanced and sufficiently high, the expected cost is $\mathcal{O}\left(\frac{1}{\lambda^2}\right)$, which *does not* depend on the network diameter.

The network bandwidth requirement for the tree update process is somewhat higher: not only does it include child-parent communication costs, but it also includes costs incurred when updating neighbors' cached presence bits.

$$
\begin{aligned}
\mathcal{B}\left(\mathcal{U}\right) &\leq k + \sum_{i=0}^{L-1} \left(3^2 - 1\right) 2^i \\
&\leq k + 8\left(2^{\lceil \log_2 k \rceil}\right) \\
&\leq k + 16k
\end{aligned}
$$

The worst-case bandwidth requirement for the tree update process is again $\mathcal{O}\left(k\right)$.

The expected bandwidth requirement is significantly lower:

$$
\begin{aligned}
\mathrm{E}[\mathcal{B}\left(\mathcal{U}\right)] &= \mathrm{E}[\mathcal{U}] + \mathrm{E}[\mathcal{B}\left(\text{neighbor-cache update}\right)] \\
&\leq \frac{(1-\lambda)}{4}\left[1 + \frac{1}{\lambda^2}\right] + \sum_{i=0}^{L-1} 8\left(2^i\right)\mu^{\left(4^i\right)} \\
&\leq \frac{(1-\lambda)}{4}\left[1 + \frac{1}{\lambda^2}\right] + 8\mu \frac{d}{d\mu}\frac{1}{1-\mu} \\
&\leq \frac{(1-\lambda)}{4}\left[1 + \frac{1}{\lambda^2}\right] + \frac{8\left(1-\lambda\right)}{\lambda^2}
\end{aligned}
$$

The expected bandwidth requirement is again $\mathcal{O}\left(\frac{1}{\lambda^2}\right)$.

Shared-Memory

The shared-memory analysis for the two-dimensional case follows the message-passing analysis in the same way that it does for the one-dimensional case. The resulting cost goes up only by a constant factor.

5.3 n-Dimensional Case

This section generalizes the analyses presented in the previous two sections. Here, we look at the behavior of the X-Tree algorithm on an n-dimensional mesh, for any n. We only examine the case of message-passing: the shared-memory costs go up by a factor of $\log l$ for search and a constant for update.

Throughout this section, the smallest number of hops d between two nodes on processors $\mathbf{P}_{i_0,i_1,\ldots,i_{n-1}}$ and $\mathbf{P}_{j_0,j_1,\ldots,j_{n-1}}$ is simply $|j_0 - i_0| + |j_1 - i_1| + \ldots + |j_{n-1} - i_{n-1}|$:

$$\begin{aligned} d &= \mathcal{D}\left(N^0_{i_0,i_1,\ldots i_{n-1}}, N^0_{j_0,j_1,\ldots,j_{n-1}}\right) \\ &= |j_0 - i_0| + |j_1 - i_1| + \ldots + |j_{n-1} - i_{n-1}| \end{aligned}$$

This is the lowest possible cost for communicating between $\mathbf{P}_{i_0,i_1,\ldots,i_{n-1}}$ and $\mathbf{P}_{j_0,j_1,\ldots,j_{n-1}}$.

When following the shortest path through the tree between $N^0_{i_0,i_1,\ldots,i_{n-1}}$ and $N^0_{j_0,j_1,\ldots,u_{n-1}}$, we have to ascend l levels:

$$\max\left(\lfloor\log_2|j_0 - i_0|\rfloor, \ldots, \lfloor\log_2|j_{n-1} - i_{n-1}|\rfloor\right) \le l \le \max\left(\lceil\log_2|j_0 - i_0|\rceil, \ldots, \lceil\log_2|j_{n-1} - i_{n-1}|\rceil\right)$$

$$\lfloor\max\left(\log_2|j_0 - i_0|, \ldots, \log_2|j_{n-1} - i_{n-1}|\right)\rfloor \le l \le \lfloor\max\left(\log_2|j_0 - i_0|, \ldots, \log_2|j_{n-1} - i_{n-1}|\right)\rfloor$$

$$\lfloor\log_{(2^n)} d\rfloor \le l \le \lceil\log_2 d\rceil$$

The exact value for l depends on how the nodes are aligned with respect to the tree.

Finally, L is the height of the tree:

$$\lfloor log_2 k\rfloor \le L \le \lceil log_2 k\rceil$$

5.3.1 Search

In this section, we show that the cost of searching the X-Tree for nearby work is n-competitive with the optimal adversary. In a k-ary n-dimensional mesh, a path from a node $N^0_{i_0,i_1,\ldots,i_{n-1}}$ to node $N^0_{j_0,j_1,\ldots,j_{n-1}}$ is established by ascending l levels in the tree from node $N^0_{i_0,i_1,\ldots,i_{n-1}}$ to node $A^l_{i_0,i_1,\ldots,i_{n-1}}$, crossing over to node $A^l_{j_0,j_1,\ldots,j_{n-1}}$ and descending back down to node $N^0_{j_0,j_1,\ldots,j_{n-1}}$. The following analysis demonstrates the competitive factor of n:

$$
\begin{aligned}
\mathcal{X}\left(N^0_{i_0,i_1,\ldots,i_{n-1}}, N^0_{j_0,j_1,\ldots,j_{n-1}}\right) &= \quad \mathcal{D}\left(N^0_{i_0,i_1,\ldots,i_{n-1}}, A^l_{i_0,i_1,\ldots,i_{n-1}}\right) \\
&\quad + \mathcal{D}\left(A^l_{i_0,i_1,\ldots,i_{n-1}}, A^l_{j_0,j_1,\ldots,j_{n-1}}\right) \\
&\quad + \mathcal{D}\left(A^l_{j_0,j_1,\ldots,j_{n-1}}, N^0_{j_0,j_1,\ldots,j_{n-1}}\right) \\
&\leq \left[1 + \sum_{x=1}^{l-1} n\left(2^{x-1}\right)\right] + n\left(2^l\right) + \left[1 + \sum_{x=1}^{l-1} n\left(2^{x-1}\right)\right] \\
&\leq n\left(2^{l-1} + 2^l + 2^{l-1}\right) \\
&\leq 4n\left(2^{l-1}\right) \\
&\leq 4n\left(2^{\lceil \log_2 d \rceil - 1}\right) \\
&\leq 4nd
\end{aligned}
$$

Therefore, the traversal cost for the X-Tree is no more than a factor of $4n$ worse than direct access through the mesh.

As is shown for the one- and two-dimensional cases, both for the X-Tree algorithm and for the optimal adversary, a search costs twice as much as a simple message send from source to destination. Therefore, the competitive factor of $4n$ holds for a search as well as for a one-way message send.

We now derive the worst-case network bandwidth requirement for a search that begins at $N^0_{i_0,i_1,\ldots,i_{n-1}}$ when the closest work is at $N^0_{j_0,j_1,\ldots,j_{n-1}}$:

$$\begin{aligned}
\mathcal{B}\left(N^0_{i_0,i_1,\ldots,i_{n-1}}, N^0_{j_0,j_1,\ldots,j_{n-1}}\right) \leq & \left(1+\sum_{x=1}^{l-1} n\left(2^{x-1}\right)\right) \\
& + n\left(2^l\right) \\
& + \left[1+\sum_{x=0}^{l-1}\left[\left(2^n\right)^{l-x}\right](n)\left(2^{x-1}\right)\right] \\
\leq & \; n\left[2^{l-1}+2^l+l\left(2^n\right)^{l-1}\right] \\
\leq & \; n\left[l+3\right]\left[\left(2^n\right)^{l-1}\right] \\
\leq & \; n\left[3+\lceil \log_2 d\rceil\right]\left[\left(2^n\right)^{\lceil \log_2 d\rceil - 1}\right] \\
\leq & \; n\left[4+\log_2 d\right] d^n
\end{aligned}$$

Although the running time for the search is $\mathcal{O}(nd)$, where d is the distance between the searcher and the nearest work, the network bandwidth consumed is $\mathcal{O}(nd^n \log d)$. Again, this is expected behavior, due to the use of a gathering process to collect threads from the entire sub-tree rooted at $A^l_{j_0,j_1,\ldots,j_{n-1}}$.

5.3.2 Presence Bit Update

The presence bit update analysis for n dimensions is similar to that for one and two dimensions. First, the bad news: the *worst-case* tree update critical-path cost can be as bad as $\mathcal{O}(nk)$. We now show why this is the case.

$$\begin{aligned}
\mathcal{U} &\leq n\left(1+\sum_{i=0}^{L-2} 2^i\right) \\
&\leq n2^{L-1} \\
&\leq n2^{\lceil \log_2 k \rceil - 1} \\
&\leq nk
\end{aligned}$$

Yet again, we find that the worst-case tree update cost is proportional to the network diameter.

As usual, the *expected* tree update cost is much lower on a machine whose workload is both balanced and sufficiently high. As before, these load conditions are expressed by *any* distribution of thread queue lengths such that the probability of non-zero queue length is at least λ, for some $\lambda : \text{Prob}\,(\text{qlen} > 0) \geq \lambda$.

$$
\begin{aligned}
\mathrm{E}[\mathcal{U}] &= \sum_{i=0}^{L-2} \mathcal{C}\,(i_0, i_1, ..., i_{n-1})\, P\,(i_0, i_1, ..., i_{n-1}) \\
&\leq n \left[2^{-1} \left(\mu^{\left[(2^n)^0\right]} - \mu^{\left[(2^n)^1\right]} \right) + 2^0 \left(\mu^{\left[(2^n)^1\right]} - \mu^{\left[(2^n)^2\right]} \right) \right. \\
&\qquad \left. + ... + 2^{L-4} \left(\mu^{\left[(2^n)^{L-3}\right]} - \mu^{\left[(2^n)^{L-2}\right]} \right) + 2^{L-3} \left(\mu^{\left[(2^n)^{L-2}\right]} \right) \right] \\
&\leq n \left(\mu 2^{-2} + \sum_{i=0}^{L-2} 2^{i-2} \mu^{\left[(2^n)^i\right]} \right) \\
&\leq \frac{n}{4} \left(\mu + \sum_{i=0}^{L-2} 2^i \mu^{\left[(2^n)^i\right]} \right) \\
&\leq \frac{n}{4} \left(\mu + \sum_{i=0}^{L-2} \left[2^n\right]^i \mu^{\left[(2^n)^i\right]} \right) \\
&\leq \frac{n}{4} \left[\mu + \sum_{j=1}^{\infty} j \mu^j \right] \\
&\leq \frac{\mu n}{4} \left[1 + \frac{1}{(1-\mu)^2} \right] \\
&\leq \frac{n\,(1-\lambda)}{4} \left[1 + \frac{1}{\lambda^2} \right]
\end{aligned}
$$

This result shows that although the worst-case tree update cost is proportional to the network diameter, on a machine whose workload is both balanced and sufficiently high, the expected cost is $\mathcal{O}\left(\frac{n}{\lambda^2}\right)$, which *does not* depend on the network diameter.

The network bandwidth requirement for the tree update process is somewhat higher; not only does it include child-parent communication costs, but it also includes costs incurred when updating neighbors' cached presence bits:

$$\begin{aligned}
\mathcal{B}\left(\mathcal{U}\right) &\leq k + \sum_{i=0}^{L-1} \left(3^n - 1\right) 2^i \\
&\leq k + \left(3^n - 1\right) \left(2^{\lceil \log_2 k \rceil}\right) \\
&\leq k + \left(3^n - 1\right) k
\end{aligned}$$

The worst-case bandwidth requirement for the tree update process is $\mathcal{O}\left(3^n k\right)$.

The expected bandwidth requirement is significantly lower on a machine workload is balanced and high enough:

$$\begin{aligned}
\mathrm{E}[\mathcal{B}\left(\mathcal{U}\right)] &= \mathrm{E}[\mathcal{U}] + \mathrm{E}[\mathcal{B}\left(\text{neighbor-cache update}\right)] \\
&\leq \frac{n\left(1-\lambda\right)}{4}\left[1 + \frac{1}{\lambda^2}\right] + \sum_{i=0}^{L-1} \left(3^n - 1\right)\left(2^i\right)\left(\mu^{\lceil (2^n)^i \rceil}\right) \\
&\leq \frac{n\left(1-\lambda\right)}{4}\left[1 + \frac{1}{\lambda^2}\right] + \left(3^n - 1\right)\mu \frac{d}{d\mu}\frac{1}{1-\mu} \\
&\leq \frac{n\left(1-\lambda\right)}{4}\left[1 + \frac{1}{\lambda^2}\right] + \frac{\left(3^n - 1\right)\left(1-\lambda\right)}{\lambda^2}
\end{aligned}$$

The expected bandwidth requirement is $\mathcal{O}\left(\frac{3^n}{\lambda^2}\right)$.

Chapter 6

Experimental Method

This chapter describes experiments carried out in the course of this research. This description covers three main areas: simulation environments, thread management algorithms and applications.

Our tests employed two simulation environments. The first, named NWO, is an accurate cycle-by-cycle simulator of the Alewife machine. NWO is the primary vehicle for Alewife system software development. The second, named PISCES, is a faster but less accurate simulator built specifically for this research. PISCES was the main data-gathering apparatus for this thesis, allowing us to simulate systems of up to 16384 processors. We extracted the parameters used to drive PISCES from simulations run using NWO, as described in Section 6.1.

A number of thread management algorithms were implemented to run on PISCES. These include the X-Tree algorithm used by **XTM**, two other combining-tree algorithms (**TTM** and **XTM-C**), two diffusion-based thread managers (**Diff-1** and **Diff-2**), two round-robin thread managers (**RR-1** and **RR-2**) and four idealized thread managers (**Free-Ideal**, **P-Ideal**, **C-Ideal-1** and **C-Ideal-2**). All of these thread managers share a single queue management discipline, which tends to increase thread locality and avoid memory overflow problems, while encouraging a uniform spread of threads around the machine. See Section 6.4 for the details of the various thread managers.

Five applications were run on PISCES under the various candidate thread managers: a two-dimensional integrator that employs an adaptive quadrature algorithm (AQ), a branch-

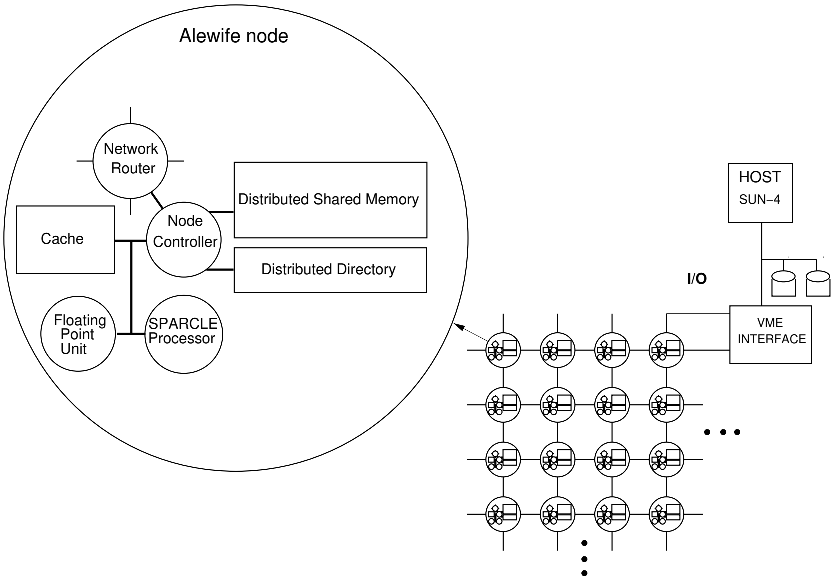


Figure 6-1: The Alewife Machine.

and-bound solver for the traveling salesman problem (TSP), doubly recursive Fibonacci (FIB), a block matrix multiply (MATMUL), and a synthetic unbalanced application (UNBAL) which tests system behavior in the face of an initially poor load distribution. Details of the various applications are given in Section 6.5.

6.1 NWO: The Alewife Simulator

Alewife is an experimental multiprocessor being developed at MIT (see Figure 6-1). Alewife is primarily a shared-memory machine, containing coherent caches and a single shared address space for all processors. Alewife also supports efficient interprocessor messages, allowing programs which use a message-passing communication style to execute as efficiently as shared-memory programs.

At the time of this writing, hardware development for the Alewife machine is nearing completion. While actual hardware is not yet available, a detailed cycle-by-cycle simulator for Alewife is the primary vehicle for system software development. This simulator, dubbed NWO, performs a cycle-by-cycle simulation of the processors, the memory system and the interprocessor communications network, all of which will eventually be present in the actual Alewife hardware (see Figure 6-2). NWO is faithful enough to the Alewife hardware that

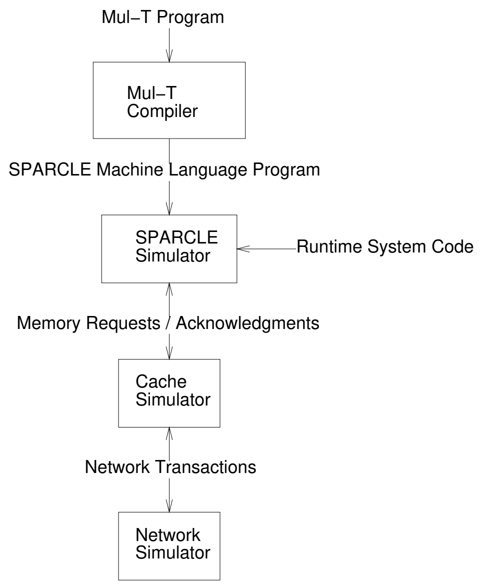


Figure 6-2: NWO Simulator Organization.

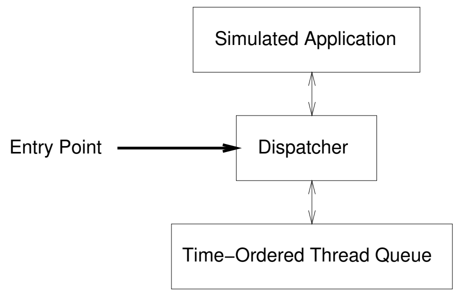


Figure 6-3: PISCES Multithreader Organization.

it has exposed many Alewife hardware bugs during the design phase.

NWO's primary drawback is its slow execution speed. NWO provides accuracy, at the cost of relatively low performance: on a SPARC-10, simulations run at about 2000 clock cycles per second. This means that a typical 64-processor simulation runs approximately two million times slower than it would on the actual hardware. For this reason, it is impossible to run programs of any appreciable size on NWO. Therefore, for the purposes of this thesis, NWO was primarily employed as a statistics-gathering tool. Parameters from NWO runs were used to drive PISCES, a faster, higher-level simulator, described in the next section.

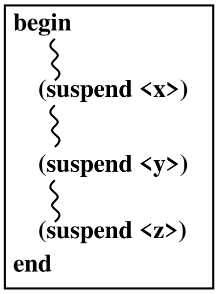


Figure 6-4: A Typical PISCES Thread.

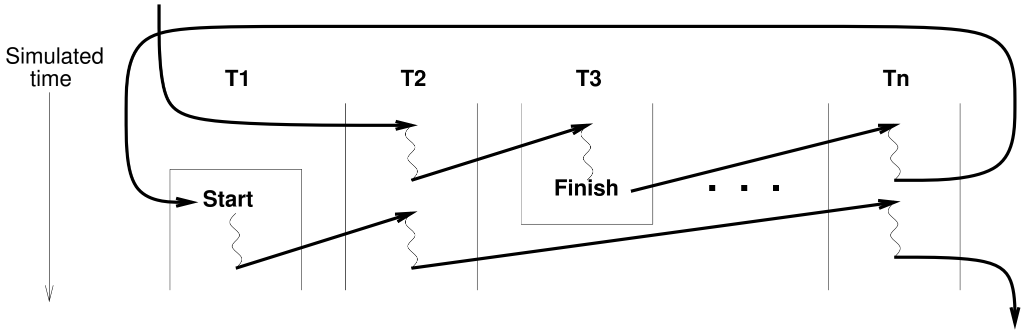


Figure 6-5: PISCES Thread Execution Order.

6.2 The PISCES Multiprocessor Simulator

The PISCES multiprocessor simulator is a general, low-overhead multiprocessor simulation system, consisting of a simple multithreading system and a machine model. The entire system is written in T [23], a dialect of LISP.

The multithreader diagrammed in Figure 6-3 supports multiple independent threads of computation executing in a common namespace. Each thread is a T program with its own execution stack. A typical thread consists of a series of blocks of code, separated by expressions of the form `(suspend <t>)`. Each `suspend` expression informs the system that the associated code block requires `t` cycles to run (see Figure 6-4). The `suspend` mechanism is the only way for a thread to move forward in time; all code run between `suspend` expressions is atomic with respect to the rest of the simulation.

PISCES threads are sorted into a time-ordered queue. The multithreader takes the first thread from the queue, runs it until it encounters the next `(suspend <t>)` expression, and re-enqueues it to execute `t` cycles later (see Figure 6-5). New threads are created by calling

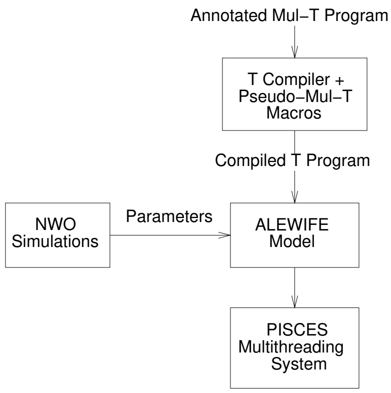


Figure 6-6: PISCES Alewife Simulation.

`(make-ready <proc> <t>)`, which creates a new thread to run `t` cycles from the current simulated time, and which will run procedure `proc` when it first wakes up. When `proc` finishes, the thread terminates.

The PISCES Alewife machine model diagrammed in Figure 6-6 is built on top of the PISCES multithreader. A simulated processor can be in one of three states:

Thread Manager A processor begins its life running the thread manager. It continues in this state until it finds an application thread to run. At this point, the thread manager associated with this processor is put aside and the processor begins to run the application thread.

Application The threads that make up the application being run on the simulated machine are implemented as PISCES threads. An application thread continues to run until it either terminates or suspends on a synchronization datatype.

Interprocessor Message Messages that are sent between simulated processors are also implemented as PISCES threads. When a simulated processor sends a message to another processor, it creates a new PISCES thread to be run on the other processor c cycles in the future, where c is the number of cycles that it takes to send a message between the two processors. At that time, the destination processor is interrupted, and the message thread is executed. The interrupt mechanism is described below.

PISCES threads run until they release control through `suspend` expressions. Furthermore, application threads can be suspended by performing operations on certain synchro-

Parameter	Description	Default Value
`Int-Ovh`	Interrupt Processing Overhead	18 cycles
`Msg-Send-Ovh`	Message Send Overhead	18 cycles
`Msg-Rcv-Ovh`	Message Receive Overhead	18 cycles
`Thd-Cre-Ovh`	Time to Create a New Thread Message	13 cycles
`Thd-Rcv-Ovh`	Time to Receive and Instantiate a Thread	67 cycles
`Thd-Enb-Ovh`	Time to Enable a Suspended Thread	14 cycles
`Thd-Load-Ovh`	Time to Load a New Thread	29 cycles
`Thd-Spnd-Ovh`	Time to Suspend a Thread	99 cycles
`Thd-Rld-Ovh`	Time to Reload a Thread	56 cycles
`Thd-Term-Ovh`	Time to Terminate a Thread	32 cycles
`Sched-Entry`	Time to Enter the Thread Manager	8 cycles
`Sched-Local`	Time to Check the Local Thread Queue	18 cycles

Table 6.1: Timing Parameters: *Obtained from NWO simulations.*

nization datatypes, described below. Since this is a uniprocessor simulation, nothing can change the state of the simulated machine while a given thread is running (between `suspend` or synchronization operations). Therefore, users of the PISCES multithreading package are encouraged to keep blocks between `suspend` calls small, in order to improve the accuracy of the simulation.

A processor can be interrupted at any time. When an interrupt message is received, the associated interrupt thread is executed at the next break. This could result in poor timing behavior in the presence of long `suspend` operations: if a thread is suspended for a long period of time when an interrupt is received, the interrupt will not be processed until the thread resumes execution and is then suspended again. For this reason, all "long" suspend operations are executed as a series of shorter suspend operations. Currently, the *suspend quantum* is ten cycles: this seems to give a fair balance between performance of the simulation and accuracy of timing behavior.

6.2.1 Timing Parameters

The PISCES Alewife simulation requires a number of timing parameters to be set. These parameters describe the timing behavior of the machine being simulated. For the purposes of this thesis, these parameters were obtained through measurements of NWO simulations. They are summarized in Table 6.1.

6.2.2 Inaccuracies

The PISCES Alewife simulation executes from one to two orders of magnitude faster than NWO. This gain in performance has a cost: PISCES simulations contain a number of inaccuracies not found in NWO simulations.

First, cache behavior is ignored unless specifically modeled in the application (see description of Cached MATMUL given below). All timing figures given in Table 6.1 assume cache hits, except in those sections of the code where the cache is guaranteed to miss, in which case local cache miss timing is assumed. Furthermore, all data accesses are free, except those whose cost is explicitly modeled through `suspend` expressions. For most of the applications described below, all data accesses are to the execution stack, which is in local memory and usually resident in the cache, thus minimizing the effect of this inaccurate behavior.

Second, the network model assumes no contention. The costs associated with a message send include fixed source and destination costs and a variable cost depending on the size of the message, the distance between source and destination, and the network speed:

$$\mathtt{MsgCost} = \mathtt{MsgSendOvh} + \mathtt{MsgRcvOvh} + [\mathtt{MsgSize} + \mathtt{Dist}(\mathtt{Src} \Rightarrow \mathtt{Dst})] \times \mathtt{NetSpeed}$$

Third, as discussed above, the behavior of `suspend` calls affects the timing behavior of the entire simulation. Blocks of code executed between suspend calls appear atomic to the rest of the simulation. Furthermore, any inaccuracies in the argument to the `suspend` associated with a block show up as inaccuracies in the running time of that thread. Finally, interrupts can only occur at *suspend quantum* boundaries. This means that if a message shows up at a processor at time t, it might not be executed until time $t + q$, where q is the suspend quantum. For all data given in Chapter 7, this quantum was set to ten cycles, which is small enough to be insignificant.

6.2.3 Synchronization Datatypes

The PISCES Alewife simulation system supports a number of synchronization datatypes, including `j-structures`, `l-structures`, and `placeholders`. These make up a subset of the datatypes provided by Mul-T [12].

`J-structures` are arrays whose elements provide write-once semantics. A read operation on a `j-structure` element is deferred until a write to that element takes place. A processor issuing such a read is suspended on a queue associated with the element. A write operation to an unwritten `j-structure` element writes a value and then frees all threads suspended while waiting for the write to take place. A write operation to a written `j-structure` element causes an error.

`L-structures` are arrays of protected data. A read performs a *lock* operation on the addressed element, returning the associated piece of data when it succeeds. A thread that attempts to read an `l-structure` cell that is already locked is suspended on a queue associated with that cell. A write performs a *unlock* operation on the addressed element, reenabling any threads suspended on that element through failed reads. A write to an unlocked `l-structure` element causes an error.

`Placeholders` are used for communication between parent and child in a `future` call. Conceptually, a `placeholder` consists of a `value`, a `queue` and a `flag`, which signifies whether the data item in the `value` slot is valid. When a `placeholder` is first created, the `flag` is set to `empty`. To change the flag to `full`, a `determine` operation must be performed on the `placeholder`. When a `future` call occurs, a child thread is created, and an associated `placeholder` is returned to the parent. If the parent tries to read the `value` associated with the `placeholder` before the `determine` operation has taken place, the parent is suspended on the `placeholder`'s `queue`. When the child thread terminates, it `determines` the value of the `placeholder`, reenabling any associated suspended threads. The semantics of a `placeholder` are very similar to those of a single `j-structure` cell.

6.3 Finding the Optimal Schedule

When evaluating the various candidate thread management algorithms, it is desirable to have a standard to compare the candidates to. The "best" schedule would be ideal for this purpose, but, as discussed in Chapter 2, the general thread management problem is NP-hard, even when the entire task graph is known. For examples of the size we're interested in, this effectively makes the optimal schedule impossible to obtain.

However, we can find a near-optimal schedule in most cases. The approach we've taken is three pronged:

1. For all applications, we know the single-processor running times and we can calculate the critical path lengths. An "optimal" T-*vs.*-p curve is derived from these two numbers as follows:
$$T_p = \max\left(\frac{T_1}{p}, T_{crit}\right)$$
where T_1 is the running time on one processor, p is the number of processors and T_{crit} is the critical path length. This yields a relatively tight lower bound on the best-case running time, and shows up in the data given in Chapter 7 under the label **Ideal**.

2. For certain applications, a near-optimal static schedule can be derived from the regular structure of the application. This approach gives a relatively tight upper bound on best-case running times when applied to the UNBAL and MATMUL applications described below.

3. For applications for which a good static schedule is not practical to obtain, we employ a more empirical approach. A number of the thread managers described below are not physically realizable. Such idealized thread managers can assume, for example, that the state of every processor's thread queue is known instantaneously on every other processor. Since we are running simulations, implementing such unrealistic thread managers is straightforward.

Together, these three approaches yield an estimate of the running time that could be achieved by an optimal schedule.

6.4 Thread Management Algorithms

A number of candidate thread management algorithms were tested for comparison against **XTM**. These algorithms can be split into two groups: realizable and unrealizable. The realizable algorithms are those that can be implemented on a real multiprocessor; the unrealizable algorithms make unrealistic assumptions that preclude their use in an actual multiprocessor. The unrealizable algorithms make it possible to gauge the effect of certain thread management costs by eliminating those costs, yielding corresponding performance figures.

6.4.1 Unrealizable Algorithms

Calculated: The **Ideal** performance figures given in Chapter 7 are not taken from any thread manager at all; they are calculated as described above in Section 6.3.

Free Idealized: **Free-Ideal** employs a single global thread queue. Push and pop operations to this queue are free, can go on concurrently. This algorithm is implemented to get a near-lower-bound on application running times, discounting communication and queue contention costs.

Idealized Producer-Driven: **P-Ideal** gives every processor instantaneous knowledge of the state of the thread queue on every other processor. When a new thread is created on a given processor, it is sent to the processor with the least amount of work on its queue, where the cost of moving the thread (which increases with distance) is added to the processor's perceived workload, so as to make distant processors less attractive than nearby processors.

Idealized Consumer-Driven 1 – ***steal-one*****:** We look at two variants of an idealized consumer-driven thread manager: **C-Ideal-1** and **C-Ideal-2**. Both versions allow every processor to have instantaneous knowledge of the state of the thread queue on every other processor. This information is used by idle, *consumer* processors, instead of busy, *producer* processors. An idle processor steals one thread from the nearest processor that has work on its queue by sending a *steal* message to that processor. If more than one processor at a given distance has work, the processor with the most work on its queue is selected.

Idealized Consumer-Driven 2 – ***steal-half*****:** **C-Ideal-2** has exactly the same behavior as **C-Ideal-2**, save for one difference concerning the number of threads moved during a *steal* operation. **C-Ideal-1** only moves one thread at a time, while **C-Ideal-2** moves *half* of the threads from the producer processor's queue to the consumer processor's queue.

6.4.2 Realizable Algorithms

Static: **Stat** is the simplest of the real thread managers. Every thread created is sent to a specific processor on the machine, as directed by the code itself. This is useful for examining static schedules, produced either by some compiler or explicitly by the user. A processor only runs those threads that are assigned to its queue.

Round-Robin-1 – *steal-one*: The round-robin thread manager comes in two forms, **RR-1** and **RR-2**, both of which use pseudo-random drafting strategies. A processor scans the queues of all other processors in a near-to-far order obtained by successively `XOR`-ing its PID with each number between 0 and $p-1$, where p is the number of processors in the machine. In this manner, each processor scans the machine in its own unique order. This order proceeds from near to far due to the mapping from PID to mesh location used by the Alewife processor [15]. In **RR-1**, when a non-empty queue is found, one thread is moved from that queue to the processor making the request.

Round-Robin-2 – *steal-half*: As was the case for **C-Ideal-1** and **C-Ideal-2**, the only difference between **RR-1** and **RR-2** concerns the number of threads that are moved. While **RR-1** moves only one thread at a time, **RR-2** moves half of the threads from producer processor to consumer processor.

Diffusion-1: Diffusion scheduling is suggested in [9]. In both **Diff-1** and **Diff-2**, every h cycles, a diffusion step is executed on each processor (the default value for h is 1000, which seems to give the best tradeoff between thread-manager overhead and effectiveness). On each diffusion step, a processor compares its queue length with those of its four nearest neighbors. For each neighbor, if the local processor's queue is longer than the neighbor's queue, r threads are sent to the neighbor, where $r = \frac{(l_0 - l_n)+3}{6}$, l_0 is the length of the local processor's queue and l_n is then length of the neighboring processor's queue.

The particular choice of the "6" in the expression $r = \frac{(l_0 - l_n)+3}{6}$ comes out of a Jacobi Over-Relaxation [3] with relaxation parameter γ set to $\frac{2}{3}$. The 3 in the numerator is present to ensure roundoff stability: for any number greater than three, a single thread can bounce back and forth between processors.

Diffusion-2: Diff-2 only differs from **Diff-1** in that the expression to determine the amount to diffuse from a processor to its neighbor is $r = \frac{(l_0 - l_n)+5}{6}$, not $r = \frac{(l_0 - l_n)+3}{6}$ as for **Diff-1**. The difference in performance can be dramatic, because **Diff-2** spreads work more uniformly around the machine. This increased performance comes at the price of roundoff stability: **Diff-2** is not completely stable in that on alternate diffusion steps, a single thread can bounce back and forth between two processors.

Single-Bit X-Tree: XTM is described in Chapter 4. One bit of presence information is maintained at each node in the tree, signifying whether there is any work available to be stolen in the subtree headed by that node.

Simple Tree: TTM is a simple tree-based thread manager similar to **XTM**, except that the tree-structure employed has no nearest-neighbor links. This lowers the cost of updating the tree, since presence bit information doesn't have to be sent to neighbors when the state of the presence bit changes. However, there is some loss of locality since tree nodes that are next to one another can be topologically distant in the tree.

Multi-Bit X-Tree: XTM-C is a multi-bit X-Tree algorithm very similar to **XTM**, except that more than one bit of presence information is maintained at each node in the tree. In order to limit the cost of updating the "weights" maintained at the tree nodes, a node informs its parent only of weight changes that cross one of a set of exponentially-spaced thresholds. In this manner, small changes in small weights are transmitted frequently while small changes in large weights, which don't matter as much, are transmitted less frequently. Furthermore, hysteresis is introduced into the system by choosing a different set of boundaries for positive and negative changes, in order to avoid repeated updates resulting from multiple small changes back and forth across a single boundary.

The node weights add a degree of conservatism to the algorithm employed by **XTM**. When a search process encounters a node with work, it doesn't always balance between the empty node and the non-empty node. Instead, a searching node balances with a neighbor only if the amount of work brought back justifies the cost of bringing the work back.

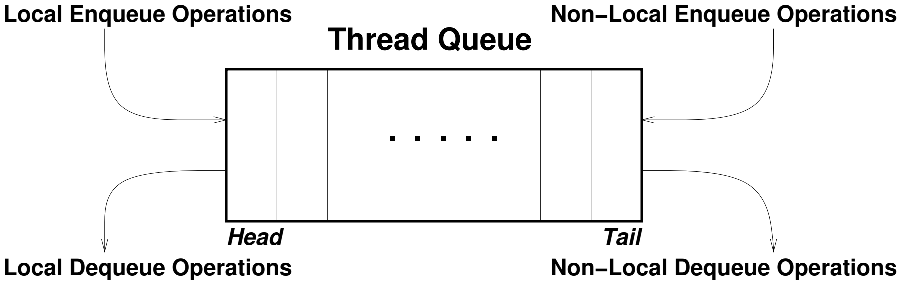


Figure 6-7: Thread Queue Management.

6.4.3 Queue Management

Certain functional aspects are shared by all of the thread management algorithms described above. In particular, they all share the same queue discipline, as pictured in Figure 6-7. When a processor creates a new thread to be put on its own local queue, it puts the thread on the head of the queue. When one or more threads are moved from one processor to another, they are taken from the tail of one queue and put on the tail of the other. Threads taken from the local queue for execution are always taken from the head of the thread queue. This causes threads to be executed in depth-first order locally, thereby minimizing memory consumption, while moving parallel threads around the machine in a more breadth-first fashion, thereby spreading work around the machine efficiently (as suggested in [8]). Furthermore, since every processor has its own thread queue, the depth-first order tended to cause threads created on a processor to remain on that processor, reducing demands on the communications network.

6.5 Applications

The candidate thread managers described above were tested on a number of example applications. These applications were chosen to fill the following requirements:

1. Any candidate application has to run for a short enough period of time so as to make PISCES simulations practical.

2. Any application chosen should be "interesting." An application is deemed to be interesting if for some machine size and problem size, near-linear speedup is possible when good thread management decisions are made. At the same time, "bad" thread management decisions should yield poor speedup for the given machine size and problem size.

3. The collection of applications chosen has to cover a range of behaviors deemed to be "typical" of dynamic applications.

The actual code for these applications can be found in Appendix B.

We first consider applications that are relatively fine-grained. Their task graphs are tree-structured; virtually all communication takes place directly between parents and children in the task tree:

Numerical Integration AQ makes use of an Adaptive Quadrature algorithm for integrating a function of two variables. This algorithm, given in [22], has a task tree whose shape is determined by the function being integrated. The particular function integrated is x^4y^4, over the square bounded by $(0.0, 0.0)$ and $(2.0, 2.0)$. Problem size is determined by the accuracy threshold: higher accuracy requires more work to achieve convergence.

Traveling Salesman TSP [18] finds the shortest path between randomly placed cities on a two-dimensional surface. In this case, problem size is determined by the number of cities scanned. The search space is pruned using a simple parallel branch-and-bound scheme, where each new "best" path length is broadcast around the machine. For problem and machine sizes tested, the detrimental effects of such broadcasts were small.

This application is unique in that the total work depends on the order in which the search tree is scanned. For the largest problem size we explored (11 cities), differences in scanning order could result in up to a factor of four in total work. However, the actual differences in total work between different runs on various machine sizes using various thread management algorithms amounted to less than a factor of two.

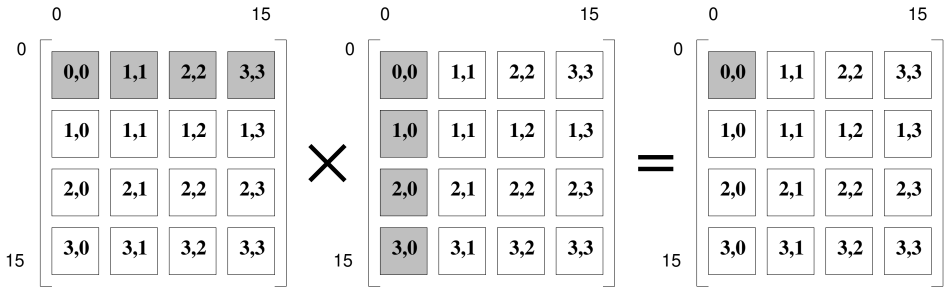


Figure 6-8: Coarse-Grained MATMUL Partitioning: *A 16-by-16 matrix times a second 16-by-16 matrix gives a third 16-by-16 matrix. Each matrix is partitioned up into 16 4-element by 4-element blocks. A single thread multiplies the entire top row of sub-blocks in the first matrix times the entire left-hand column of sub-blocks in the second matrix to calculate sub-block 0,0 of the destination matrix.*

The **Ideal** figures for this application were calculated assuming that the minimal search tree was scanned.

Fibonacci FIB(n) calculates the nth Fibonacci number in an inefficient, doubly-recursive manner. In this case, n determines the problem size.

Other applications in the test suite have more specific purposes.

Matrix Multiply All of the applications described above have a very limited communication structure. In order to test machine behavior for applications that contain more communication, a blocked matrix multiply application was included in the test suite. Four variations on the basic MATMUL were tried: coarse-grained and cached, fine-grained and cached, coarse-grained and uncached, and fine-grained and and uncached.

The cached versions simulated full-mapped caches [4]. The uncached versions were tested in order to separate out the effect of caching on the application from the thread managers' effects.

Two partitioning strategies were employed for this application, as pictured in Figures 6-8 and 6-9. The coarse-grained partitioning strategy takes advantage of locality inherent to the application. The fine-grained strategy potentially loses some of this locality, but gives the thread managers more flexibility by creating more than one thread per processor.

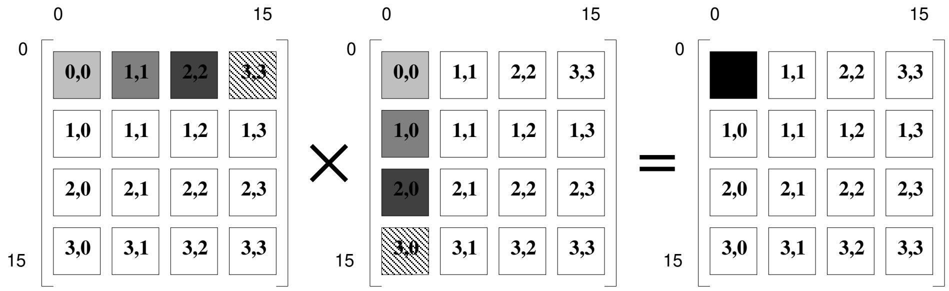


Figure 6-9: Fine-Grained MATMUL Partitioning: *A 16-by-16 matrix times a second 16-by-16 matrix gives a third 16-by-16 matrix. Each matrix is partitioned up into 16 4-element by 4-element blocks. One thread multiplies sub-block 0,0 of the first matrix with sub-block 0,0 of the second matrix to get a partial result for sub-block 0,0 of the destination matrix; another thread multiplies sub-block 0,1 of the first matrix with sub-block 1,0 of the second matrix to get another partial result for sub-block 0,0 of the destination matrix; and so on. The partial results for each destination sub-block are added together to achieve a final value; consistency is ensured using* `l-structures` *to represent the destination sub-blocks.*

In all cases, the data was distributed around the machine in the obvious way, with the upper left-hand corner of each matrix residing on the upper left-hand processor, and so on.

UNBAL UNBAL is a synthetic application whose purpose is to test the "impulse response" of a thread manager. In this application, a number of fixed-length, non-communicating threads are made to appear on one processor in the system. The program terminates when the last thread completes. This test gives some insight into a thread manager's behavior with a load that is initially severely unbalanced. In all test cases , each thread ran for 500 cycles.

Application	T_1	T_{crit}	No. Threads	Avg. Grain Size
FIB(15)	652,439	4,495	1,219	535
FIB(20)	7,244,462	6,110	13,529	535
FIB(25)	80,350,904	7,725	150,889	533
AQ(0.5)	437,191	11,725	309	1,415
AQ(0.1)	2,141,754	13,480	1,513	1,416
AQ(0.05)	4,169,108	13,480	2,943	1,417
AQ(0.01)	20,201,061	16,990	14,269	1,416
AQ(0.005)	43,062,592	16,990	30,417	1,416
AQ(0.001)	213,632,152	20,020	150,049	1,424
TSP(8)	3,635,870	4,760	$\approx$ 4,000	900
TSP(9)	12,613,984	5,605	$\approx$ 15,000	850
TSP(10)	41,708,532	6,500	$\approx$ 50,000	830
TSP(11)	326,034,362	7,445	$\approx$ 400,000	810
UNBAL(1024)	601,451	587	1,024	587
UNBAL(4096)	2,404,715	587	4,096	587
UNBAL(16384)	9,617,771	587	16,384	587
UNBAL(65536)	38,469,995	587	65,636	587

Table 6.2: Running Characteristics for Applications in the Test Suite: T_1 *is the running time in cycles on one processor.* T_{crit} *is the running time in cycles of the critical path. This is how long the program would take to run on an infinitely large multiprocessor with no communication or thread management overheads. The Average Grain Size is the average running time of the threads in the application, in cycles.*

6.5.1 Application Parameters

Table 6.2 gives the running characteristics of the various applications in the test suite for a range of application sizes.

In the next chapter, we describe the results of simulating these applications. We try to use those results to gain some insight into the behavior of the various thread management algorithms.

Chapter 7

Results

In this chapter, we present experimental results obtained using the PISCES simulator. In doing so, we attempt to demonstrate a number of points about the thread management problem in general, and about tree-based algorithms in particular.

For each application and problem size, we first identify a "region of interest" of machine sizes. If a machine is in this range, it is large enough with respect to the application so that thread management is not trivially easy, but small enough to make it possible for an incremental increase in machine size to yield a significant decrease in running time. For most of the rest of this chapter, we will only look at results that fall into that region.

We then show that the tree-based algorithms we have developed are competitive with a number of unrealizable "ideal" thread managers. The different idealized managers ignore different costs inherent to the thread-management task in order to identify the effects those costs have on overall performance. The most radical idealization, **Free-Ideal**, pays no communication or contention costs at all by scheduling threads on a single contention-free queue with zero thread enqueue and dequeue costs. In most cases, the tree-based algorithms get performances that are within a factor of three from **Free-Ideal**. The other idealized managers are usually within a factor of two of **Free-Ideal**.

A comparison of realizable algorithms then shows that the tree-based algorithms we have developed are competitive with simpler algorithms on machines with 256 or fewer processors, and that for larger machines, the tree-based algorithms yield significant performance gains over the simpler algorithms. In particular, because of their simplicity, the Round-Robin algorithms perform best of all real algorithms on small machines, but on larger machines, their performance suffers. The diffusion algorithms, on the other hand, seem to perform marginally worse than the others on machines with 256 or fewer processors; as machine size is increased, **Diff-1** and **Diff-2** perform *very* poorly with respect to the others.

When a processor from which threads can be stolen is located, consumer-based thread managers have a choice of policies when determining how much work to steal. The two choices we examine are *steal-one* and *steal-half*. These options differentiate **C-Ideal-1** and **C-Ideal-2** thread managers from each other, as well as **RR-1** and **RR-2**. We were interested in the effects of this policy choice while **XTM** was under development because **XTM** implicitly follows the *steal-half* policy: it always attempts to evenly balance the workload between two branches of a tree when moving work from one branch to the other.

We then compare the three candidate tree-based algorithms with each other. For the Alewife parameter set, we find that **TTM** (no nearest-neighbor links) performs better than **XTM**. **XTM-C** always performs poorly, despite the theoretical prediction of optimal behavior, for two reasons:

1. The work estimates maintained at the tree nodes can be inaccurate, due to time delays inherent to the update process, inaccuracies built into the system to lower update costs, and, most importantly, the incorrect assumption that all threads represent the same amount of work.

2. Maintaining work estimates in the tree carries significantly higher overhead than maintaining one-bit presence information. This added overhead results in correspondingly lower performance.

It is interesting to see what happens when the processor speed is increased with respect to the network speed. As the processor speed is increased with respect to the speed of the communications network, effects that previously showed up on large machines running large problems begin to appear on smaller machines running smaller problems. Furthermore, on machines with faster processors, the locality gains inherent in **XTM** become more important, and **XTM**'s performance surpasses that of **TTM**. This will be especially relevant if current trends in technology continue, in which processor performance is going up faster than network performance. In addition, cutting the processor cycle time gives us an inexpensive way of investigating "large machine" behavior without paying for it in simulation cycles.

Finally, we look at MATMUL, an application that demonstrates strong static data-task locality. When caches are simulated, performance depends heavily on good partitioning. The finely partitioned version fails to keep data locality within a single thread; in this case, none of the dynamic thread managers can recapture that locality. Conversely, the coarsely partitioned case keeps more accesses to the same data within each thread. For the coarsely partitioned version, the tree-based thread managers perform very nearly as well as any of the idealized dynamic managers, and almost as well as a good statically mapped version.

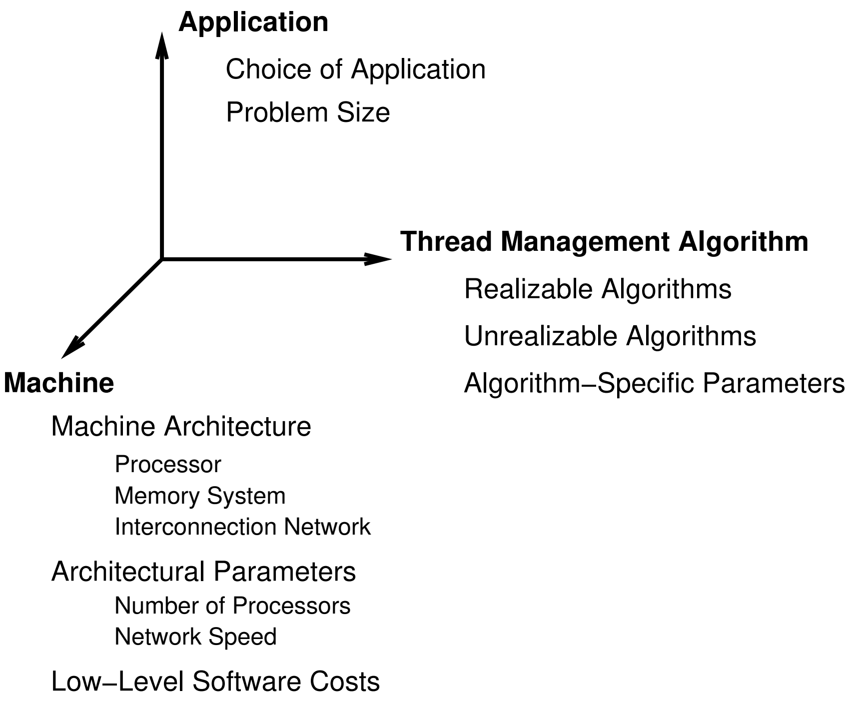


Figure 7-1: Experimental Parameters.

7.1 Parameters

When examining the performance of thread management algorithms, there are many possible parameters that can be varied, as illustrated in Figure 7-1. We separate simulation parameters into three major groups: Machine, Thread Management Algorithm and Application. In the following subsections, we discuss each major group in turn. We also classify the parameters according to whether they will remain fixed or be varied.

7.1.1 Machine Parameters

This section discusses the various architectural parameters that go into a PISCES simulation.

Machine Architecture:

Processor

Memory System

Interconnection Network

Machine Size p: The number of processors in the machine.

Network Speed t_n: The time, in cycles, that it takes one flit to travel from one switch to the next.

Low-Level Software Costs: A number of low-level activities such as thread creation, thread suspension, etc. carry costs that are independent of the thread management algorithm.

All of the above parameters were taken directly from the Alewife machine. All of these parameters except p and t_n are fixed throughout the simulations. Low-level software is assumed to be that of the prototype Alewife runtime system; the corresponding overheads were measured from NWO runs, and are given in Table 6.1.

The two machine parameters that are varied are p and t_n. We are interested in running programs on machines of various sizes, so we vary p in order to study the behavior of different thread management algorithms as machines become large. We use t_n in the same manner: one way of simulating a "large" machine is to increase interprocessor communication times. Increased t_n gives the impression of increased machine size without taking correspondingly greater time to simulate. Furthermore, as the state of the prevailing technology advances, the trend is moving towards faster and faster processors. Communication latencies are already near the speed of light and can not be reduced very far from current levels. Therefore, as processors continue to improve, we expect t_n to show a corresponding increase.

7.1.2 Thread Management Parameters

In Chapter 6, we described a number of candidate thread management algorithms to be compared against one another. These were split into two groups: realizable and unrealizable. The applications listed above were each run using each of the thread managers, repeated here for completeness:

Unrealizable

Ideal: Optimal case, calculated based on single-processor running time and critical path calculations.

Free-Ideal: Simulates a single zero-cost, zero-contention thread queue.

P-Ideal: Simulates free instantaneous knowledge of the state of all processors' thread queues. Threads are moved around by the processors that create them.

C-Ideal-1: Simulates free instantaneous knowledge of the state of all processors' thread queues. Threads are moved around by idle processors, using the "Steal-One" policy.

C-Ideal-2: Same as **C-Ideal-1** except that the "Steal-Half" policy is used.

Realizable

RR-1: A simple round-robin thread manager in which each processor scans every other processor for work, using the "Steal-One" policy.

RR-2: Same as **RR-1** except that the "Steal-Half" policy is used.

Diff-1: Diffusion-based thread manager with no instabilities.

Diff-2: Diffusion-based thread manager with a small instability; performs better than **Diff-1**.

XTM: X-Tree algorithm as described in Chapter 4.

TTM: Same as **XTM** except that no nearest-neighbor links are used: threads can only migrate between subtrees that share a parent.

XTM-C: Similar to **XTM** except that more accurate work estimates are maintained at the nodes. **XTM** maintains a single-bit work estimate (present or absent); this variant uses a multi-bit work estimate to determine whether the cost of a balance operation between two nodes justifies the expected gains.

Stat: For certain applications, a near-optimal static schedule can easily be produced. The associated thread management algorithm does nothing but run the threads that are sent to each processor in a statically determined fashion.

7.1.3 Application Parameters

Data were taken for a number of applications and problem sizes, as described in Chapter 6. We list the chosen applications and corresponding problem sizes below. For a list of application-specific characteristics, see Table 6.2.

Application

AQ: Adaptive quadrature integration.

TSP: Traveling salesman problem.

FIB: Doubly-recursive Fibonacci.

UNBAL: Synthetic "unbalanced" application.

MATMUL: Matrix Multiply.

Problem Size

AQ: Sensitivity for convergence: ranges from 0.5 to 0.001.

TSP: Number of cities in tour: ranges from 8 to 11.

FIB: Calculates the n^{th} Fibonacci number: ranges from 15 to 25.

UNBAL: Number of threads: ranges from 1024 to 65536.

MATMUL: Matrix size n ($[n \times n] \times [n \times n] = [n \times n]$): ranges from 16 to 64 .

Application	Problem Size	p Range	Managers
AQ	0.5	1–16384	all but **Stat**
	0.1	1–16384	
	0.05	1–16384	
	0.01	1–16384	
	0.005	1–16384	
	0.001	1–16384	
FIB	15	1–16384	all but **Stat**
	20	1–16384	
	25	1–16384	
TSP	8	1–4096	all but **Stat**
	9	1–4096	
	10	1–4096	
	11	1–4096	
UNBAL	1024	1–16384	all
	4096	1–16384	
	16384	1–16384	
	65536	1–16384	
MATMUL (coarse, cached)	16	1–64	all but **Ideal** and **XTM-C**
	32	1–256	
	64	1–1024	
MATMUL (coarse, uncached)	16	1–64	all but **Ideal** and **XTM-C**
	32	1–256	
	64	1–1024	
MATMUL (fine, cached)	16	1–64	all but **Ideal** and **XTM-C**
	32	1–256	
	64	1–1024	
MATMUL (fine, uncached)	16	1–64	all but **Ideal** and **XTM-C**
	32	1–256	
	64	1–1024	

Table 7.1: Experiments Performed ($t_n = 1$): *In all cases,* **RR-1** *and* **RR-2** *were tested for* $p \leq 4096$*,* **Diff-1** *and* **Diff-2** *were tested for* $p \leq 1024$ *and Coarse Uncached MATMUL with* **Diff-1** *and* **Diff-2** *was tested for* $p \leq 256$*. For UNBAL(16),* **P-Ideal**, **C-Ideal-1** *and* **RR-1** *were tested for* $p \leq 1024$*; for UNBAL(32) and for UNBAL(64), they were tested for* $p \leq 256$*. For UNBAL(1024),* **Stat** *was tested for* $p \leq 1024$*; for UNBAL(4096),* **Stat** *was tested for* $p \leq 4096$*.*

7.2 Experiments

For each application, data were taken over a range of machine sizes, problem sizes, ratios of processor speed to network speed, and thread management algorithms. Tables 7.1 and 7.2 describe the resulting (nearly complete) cross product. For a listing of raw experimental results, see Appendix C.

Note that in all cases, **RR-1** and **RR-2** were only tested for machine sizes up to 4096 processors, and **Diff-1** and **Diff-2** were only tested for machine sizes up to 1024 processors. These thread managers performed poorly on large machines; consequently, simulations of machines larger than these maximum sizes took too long to be practical.

Application	Problem Size	t_n	p Range	Managers
AQ	0.01	2	1–16384	all but **Stat**
		4	1–16384	
		8	1–16384	
		16	1–4096	
		64	1–4096	
FIB	20	2	1–16384	all but **Stat**
		4	1–16384	
		8	1–16384	
		16	1–4096	
		64	1–4096	all but **XTM-C** and **Stat**
TSP	10	2	1–4096	all but **Stat**
		4	1–4096	
		8	1–4096	
		16	1–4096	
		64	1–4096	
UNBAL	16384	2	1–16384	all
		4	1–16384	
		8	1–16384	
		16	1–4096	
		64	1–4096	
MATMUL (coarse, cached)	64	2	1–1024	all but **Ideal** and **XTM-C**
		4	1–1024	
		8	1–1024	
MATMUL (coarse, uncached)	64	2	1–1024	all but **Ideal** and **XTM-C**
		4	1–1024	
		8	1–1024	
MATMUL (fine, cached)	64	2	1–1024	all but **Ideal** and **XTM-C**
		4	1–1024	
		8	1–1024	
MATMUL (fine, uncached)	64	2	1–1024	all but **Ideal** and **XTM-C**
		4	1–1024	
		8	1–1024	

Table 7.2: Experiments Performed (Variable t_n): *For each application, a suitable problem size was selected; for that problem size, t_n was then varied. In all cases,* **RR-1** *and* **RR-2** *were tested for $p \leq 4096$,* **Diff-1** *and* **Diff-2** *were tested for $p \leq 1024$ and Coarse Uncached MATMUL with* **Diff-1** *and* **Diff-2** *was tested for $p \leq 256$. Furthermore, for UNBAL(64),* **P-Ideal**, **C-Ideal-1** *and* **RR-1** *were tested for $p \leq 256$.*

7.3 Machine Size and Problem Size - "Regions of Interest"

For each application and problem size, for the purpose of evaluating thread management algorithms, there exists a "region of interest" of machine sizes. Figures 7-2 through 7-5 display performance curves for a number of applications. All graphs in this chapter are log-log plots of machine performance (cycles) against p (number of processors) for a number of different thread managers. Virtually all of these curves have the same general shape: for small machine sizes, they are nearly linear (running time is inversely proportional to p); for large numbers of processors, they are nearly flat (running time stays constant with increasing p). Somewhat inaccurately, we term this flat region the *saturation* region for a given performance curve.

For a given application and problem size, a machine size is "interesting" if near-linear speedup is possible, but not trivial, to obtain, on a machine of that size. More specifically, for a given application and problem size, a machine size is in the region of interest if it is large enough so that thread management is not trivially easy, but small enough to make it possible for an incremental increase in machine size to yield a significant decrease in running time for *some* thread management algorithm. The region of interest is that range of machine sizes for which all performance curves of interest undergo the transition from linear to saturated. Good thread managers achieve nearly linear performance throughout most of the region, only reaching saturation towards the right hand size of the region (larger p). Bad thread managers, on the other hand, saturate near the left hand side of the region (smaller p). For most of the rest of this chapter, we will only look at results for machine sizes that are in this region.

Figures 7-2 through 7-5 give the regions of interest for AQ, FIB, TSP and UNBAL. In addition to displaying five performance curves, each graph contains two vertical dotted lines, which mark the range of p that makes up the region of interest. Notice how the region of interest moves to the right (larger machines) as problem size increases. We give the entire progression for AQ (see Figure 7-2), and the endpoints of the progressions for FIB (see Figure 7-3), TSP (Figure 7-4) and UNBAL (Figure 7-5). The region of interest data are summarized in Table 7.3.

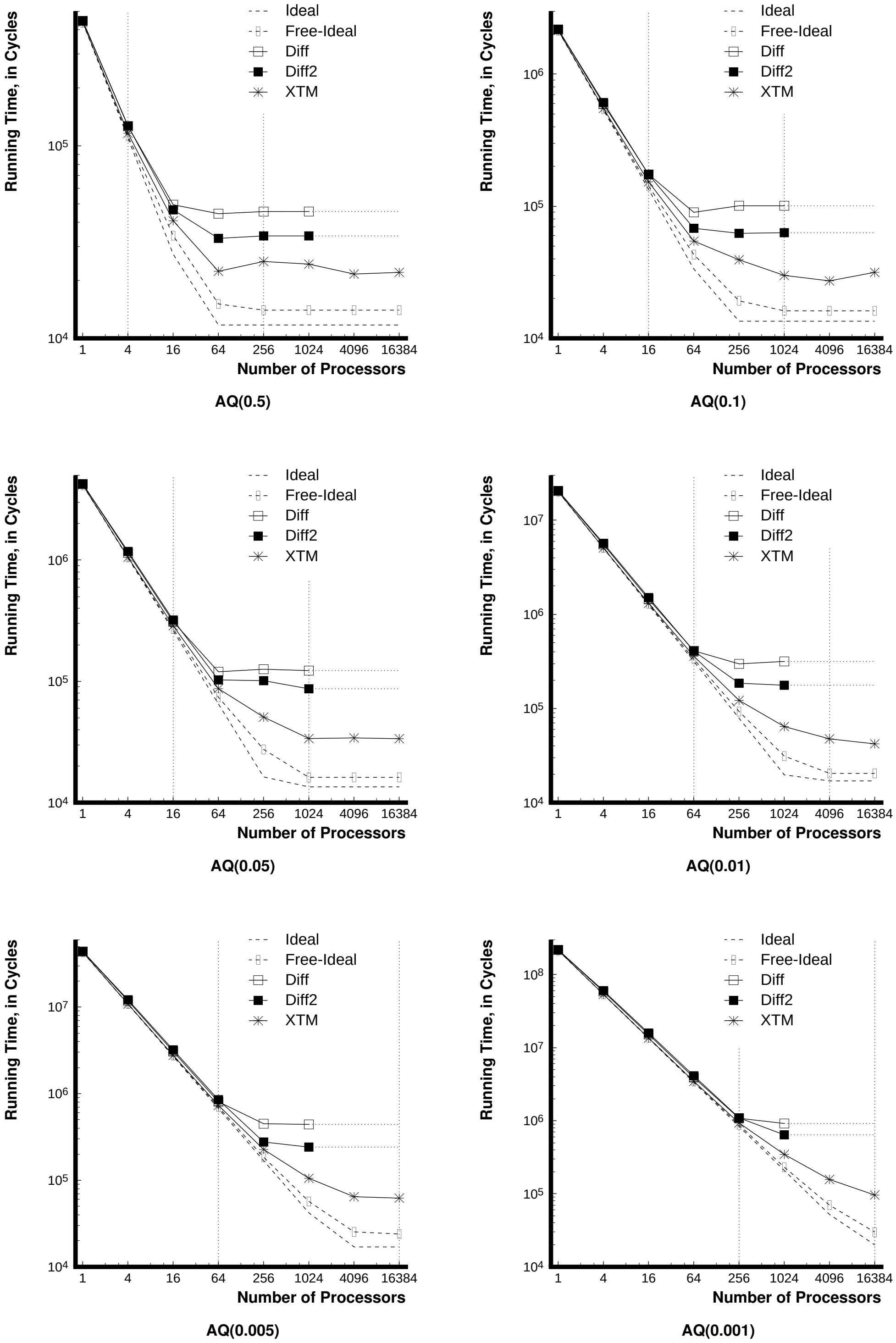


Figure 7-2: *AQ: Regions of Interest.*

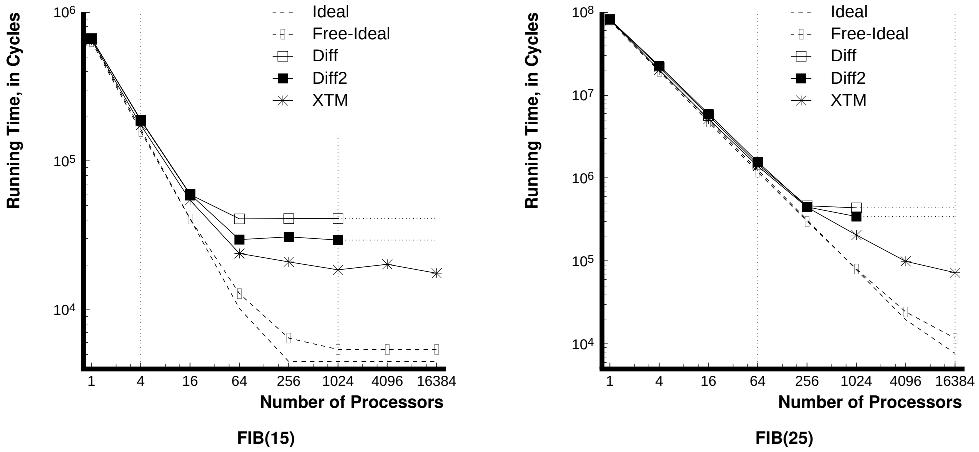


Figure 7-3: *FIB: Regions of Interest.*

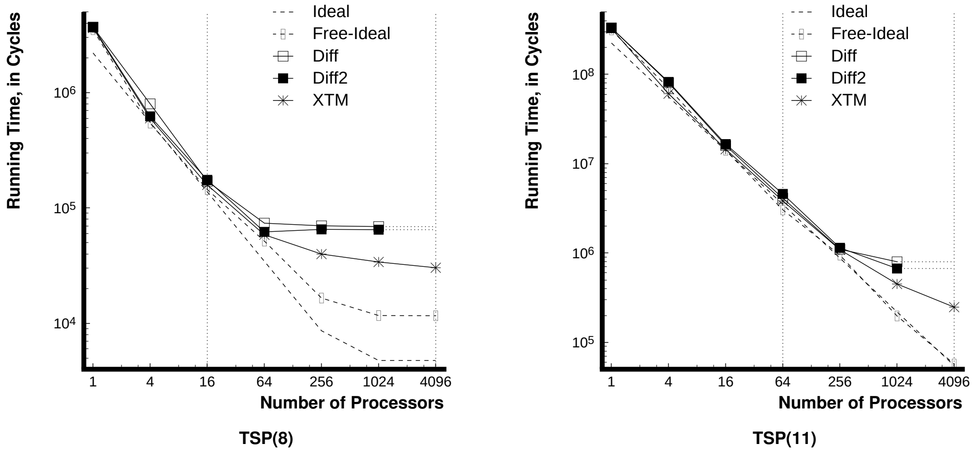


Figure 7-4: *TSP: Regions of Interest.*

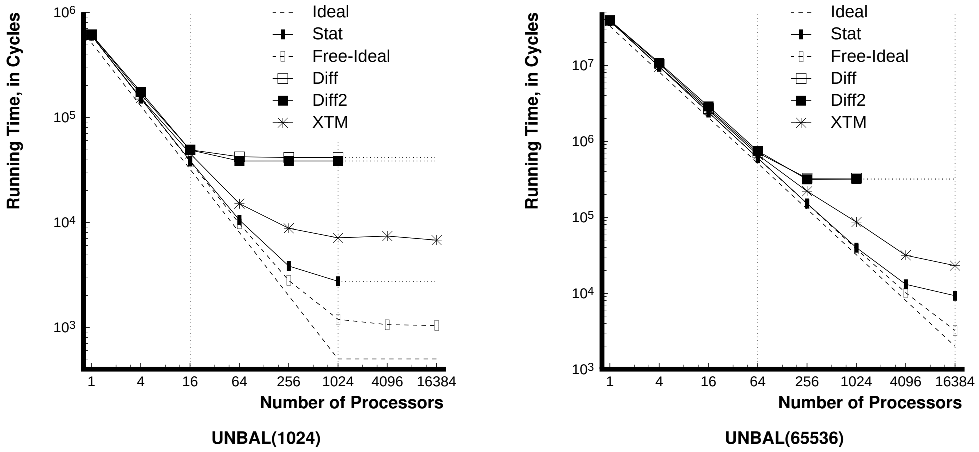


Figure 7-5: *UNBAL: Regions of Interest.*

Application	Argument	T_1	n_t	p_{int}
AQ	0.5	437,191	1,219	4–256
	0.1	2,141,754	1,513	16–1024
	0.05	4,169,108	2,943	16–1024
	0.01	20,201,061	14,269	64–4096
	0.005	43,062,592	30,417	64–16384
	0.001	213,632,152	150,049	256–16384
FIB	15	652,439	309	4–1024
	20	7,244,462	13,529	16–4096
	25	80,350,904	150,889	64–16384
TSP	8	3,635,870	$\approx$ 4,000	16–4096
	9	12,613,984	$\approx$ 15,000	64–4096
	10	41,708,532	$\approx$ 50,000	64–4096
	11	326,034,362	$\approx$ 400,000	64–4096
UNBAL	1024	601,451	1,024	16–1024
	4096	2,404,715	4,096	16–4096
	16384	9,617,771	16,384	64–16384
	65536	38,469,995	65,636	64–16384

Table 7.3: Regions of Interest: T_1 *is the running time in cycles on one processor.* n_t *is the number of threads needed for a given run.* p_{int} *is the range of* p *that makes up the region of interest. Note that the upper limit of a given region of interest is limited by the maximum number of processors on which that particular experiment was run.*

The data presented in these figures leads us to the following conclusions:

1. In all cases, to the left of the region of interest, there is very little difference between the various thread management algorithms shown. In particular, **XTM** performs as well as the others in this region. This is not surprising, since to the left of the region of interest, any thread management algorithm should do pretty well as long as it doesn't add much overhead.

2. In the region of interest, the quality of the various thread managers becomes apparent. In this region, for large applications, "good" thread managers (*e.g.*, **XTM**) achieve near-linear speedup over most of the range, while "bad" thread managers (*e.g.*, **Diff-1** and **Diff-2**) perform poorly, with run times that remain constant or increase as p increases.

3. To the right of the region of interest, all thread managers reach saturation. In some cases, adding processors actually decreases performance; in all cases, adding processors does not yield significant performance gains.

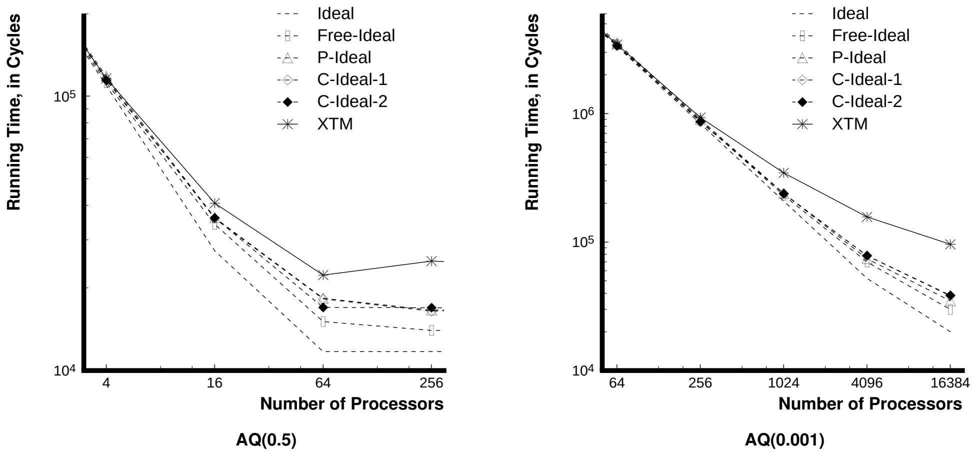


Figure 7-6: *AQ:* **XTM** *vs.* **P-Ideal**, **C-Ideal-1** *and* **C-Ideal-2**.

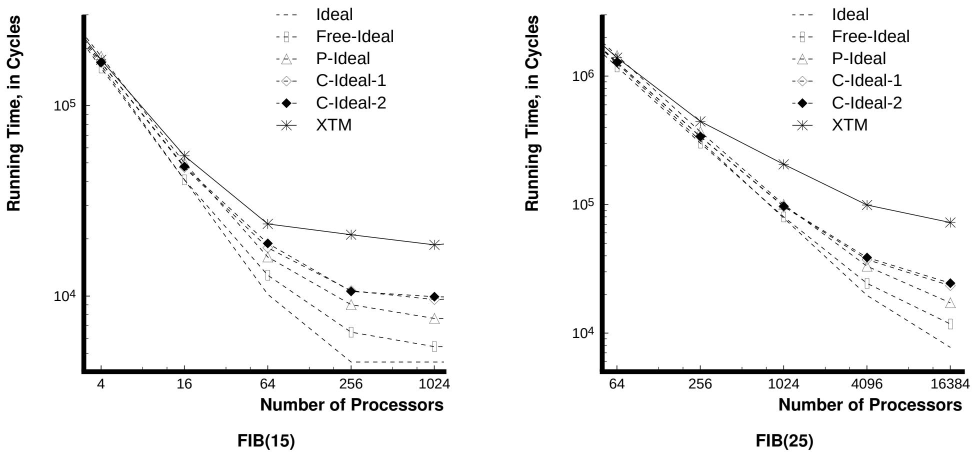


Figure 7-7: *FIB:* **XTM** *vs.* **P-Ideal**, **C-Ideal-1** *and* **C-Ideal-2**.

7.4 Comparing Tree-Based Algorithms to Unrealizable Algorithms

The tree-based algorithms we have developed exhibit performance that is competitive with a number of unrealizable "ideal" thread managers. The most idealized of these is **Free-Ideal**. This manager pays none of the communication or contention costs inherent to thread management, using a single contention-free queue with zero thread enqueue and dequeue costs. This gives a lower bound on the achievable running time when there are no inter-thread dependencies. For all but one of the applications tested here, such dependencies do exist, but a simple heuristic seems to get near optimal performance in all cases: run threads

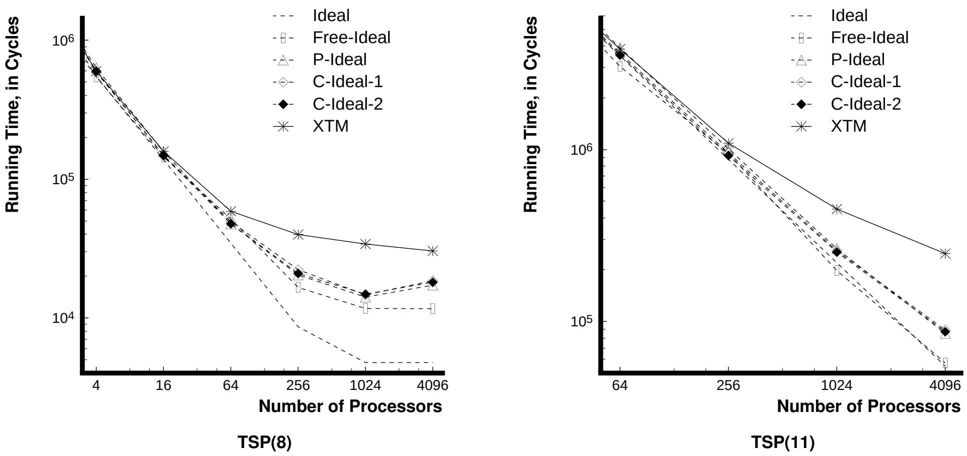


Figure 7-8: *TSP:* **XTM** *vs.* **P-Ideal**, **C-Ideal-1** *and* **C-Ideal-2**.

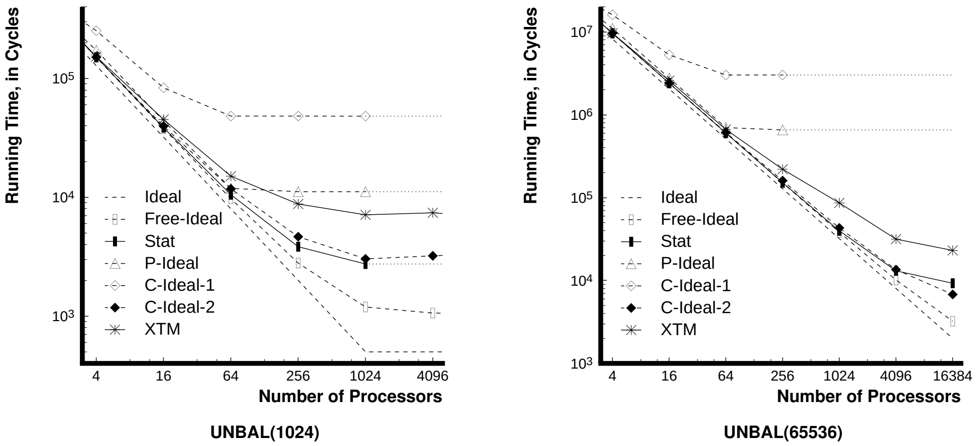


Figure 7-9: *UNBAL:* **XTM** *vs.* **P-Ideal**, **C-Ideal-1** *and* **C-Ideal-2**.

on a single processor in a depth-first fashion and put all reenabled threads on the head of the running queue.

The other idealized thread managers pay communication costs for moving threads and values around, but have "free" information on the state of the system at all times. The producer-oriented version (**P-Ideal**) uses this information to "push" newly created threads from the creating processor to destinations chosen by distance and queue length. The consumer-oriented versions (**C-Ideal-1** and **C-Ideal-2**) "pull" threads from the nearest non-empty queue. When **P-Ideal** sends a thread to a queue, it makes a note of the fact so that other processors realize that the queue is about to get longer and act accordingly. When **C-Ideal-1** or **C-Ideal-2** decides to take one or more threads off a queue, it makes a similar note so that other processors know that those threads are "spoken for."

The relation between **P-Ideal** to **C-Ideal-1** and **C-Ideal-2** is that of eager to lazy. When the delay between when a thread is created and when it is sent to its final destination is most relevant to achieving good performance, **P-Ideal** does well. However, in general, the later a decision is made, the more information is available and the better the choice. Therefore if the inaccuracy of choice inherent to **P-Ideal** is a major factor, **C-Ideal-1** and **C-Ideal-2** will perform better. For the applications tested, dispatch time was slightly more important than accuracy of choice, so **P-Ideal** usually beat out **C-Ideal-1** and **C-Ideal-2**by a slight margin. Figure 7-9 is the exception, again by a small margin.

In all cases, **Free-Ideal** achieved the best performance, followed by **P-Ideal** and then **C-Ideal-1** and **C-Ideal-2**. On large numbers of processors, The idealized thread managers outperformed **XTM** by a factor of 1.5 to four. It seems that the primary advantage the idealized managers have over **XTM** stems from the availability of free, completely up-to-date global information; the real thread managers only have the information that is made available to them. This information is in general both late and inaccurate.

From the data presented in Figures 7-6 through 7-9, we derive the following conclusions:

1. There doesn't seem to be much difference between eager and lazy decision-making for the applications we tested.

2. The costs inherent to the collection and dissemination of global information can be prohibitively high.

Finally, note that in Figure 7-9, the **P-Ideal** and **C-Ideal-1** results saturate very early in the region of interest. This is due to a serialization inherent to those algorithms (see Section 7.6).

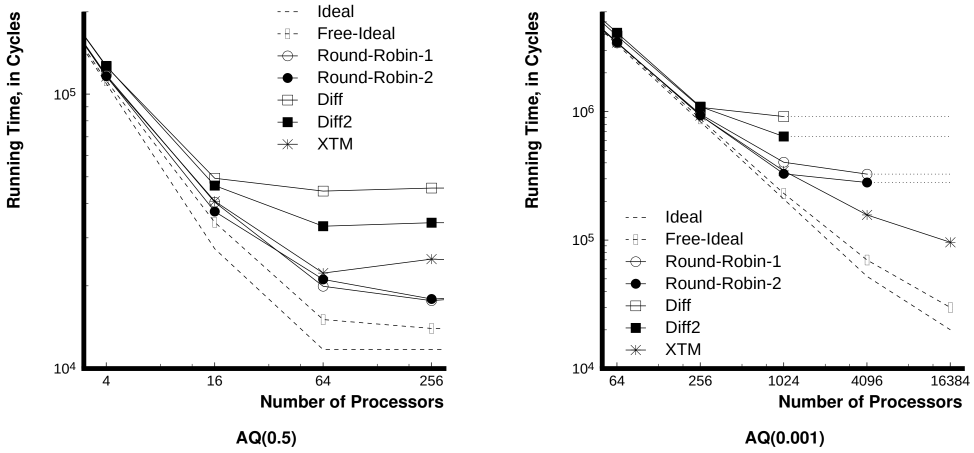


Figure 7-10: *AQ:* **XTM** *vs.* **Diff-1**, **Diff-2**, **RR-1** *And* **RR-2**.

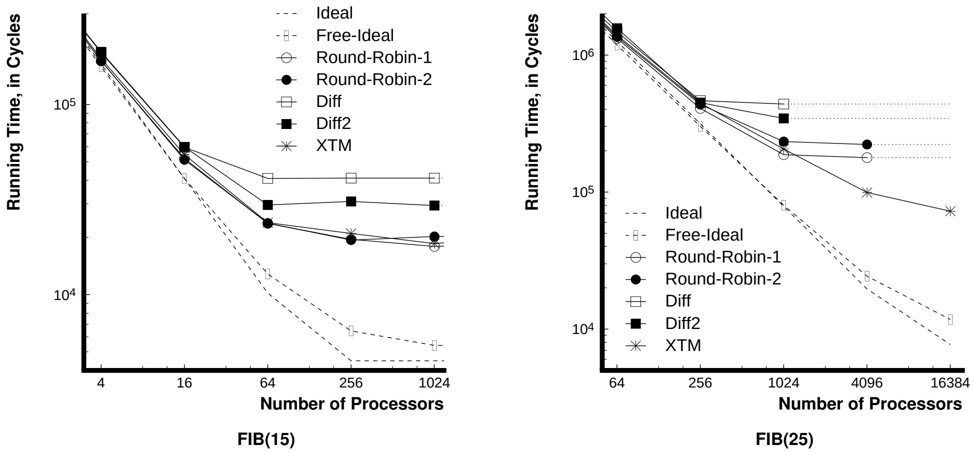


Figure 7-11: *FIB:* **XTM** *vs.* **Diff-1**, **Diff-2**, **RR-1** *And* **RR-2**.

7.5 Comparing Tree-Based Algorithms to Other Realizable Algorithms

A comparison of realizable algorithms speaks favorably for **XTM**. Figures 7-10 through 7-13 compare **XTM**, **Diff-1**, **Diff-2**, **RR-1** and **RR-2**, using **Ideal** and **Free-Ideal** as baselines. Note that for small problem sizes, whose regions of interest cover relatively small machine sizes, there is little to choose from between the various thread managers. However, for larger problem sizes, a significant performance difference between the thread managers begins to appear.

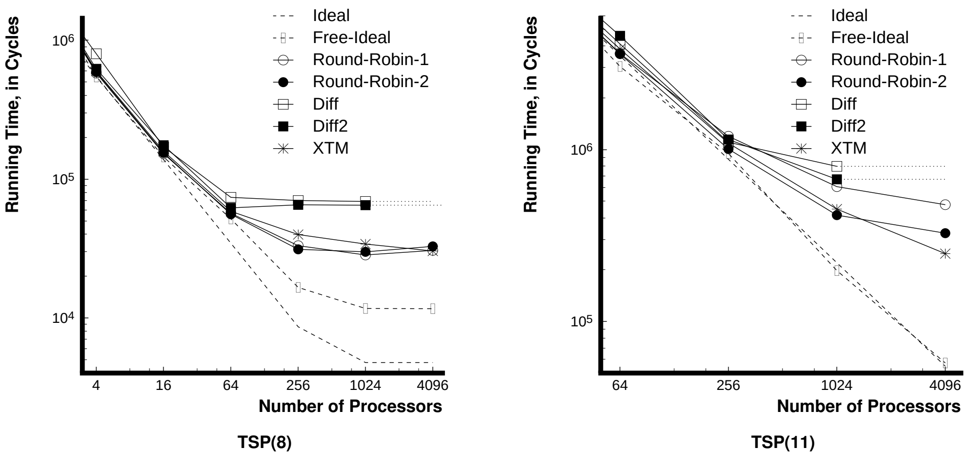


Figure 7-12: *TSP:* **XTM** *vs.* **Diff-1**, **Diff-2**, **RR-1** *And* **RR-2**.

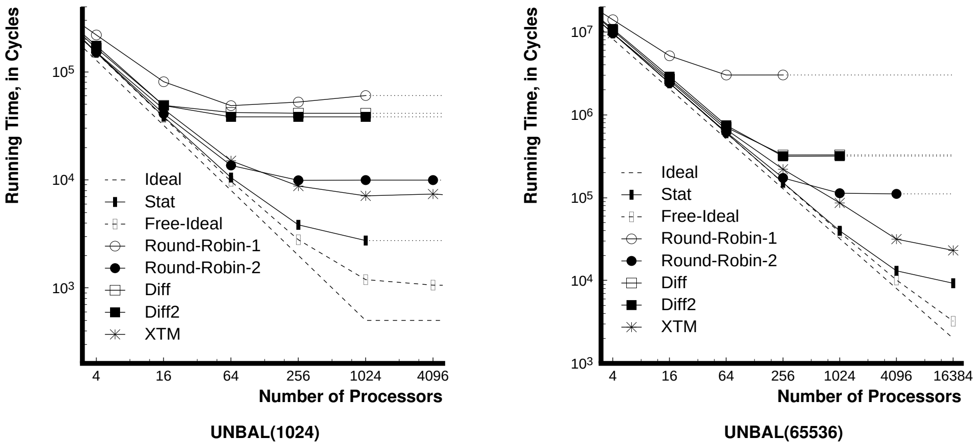


Figure 7-13: *UNBAL:* **XTM** *vs.* **Diff-1**, **Diff-2**, **RR-1** *And* **RR-2**.

RR-1 and **RR-2** run well on small machines. Their simple structure requires no additional overhead on any thread creation or consumption activities; no distributed global data structures need to be updated. As long as communication costs remain low, Round-Robin is fine, but as machine size increases, **RR-1** and **RR-2** place requirements on the communication system that it can't fill. In particular, if when running a Round-Robin thread manager, a processor queries five other processors before finding work, then as long as that query time is small compared to the thread loading, running and termination times, the Round-Robin manager will achieve good performance. As machine sizes increase, the latency of each query goes up, and eventually the query time dominates the performance.

Diff-1 and **Diff-2** perform slightly worse than the others on small machines, and very poorly on large machines. There are several reasons for this. Most important, the need for algorithmic stability in the relaxation leads to a "minimum slope" in the load on adjacent processors: when the load on two neighboring processors differs by two or less, no exchange takes place on a relaxation step. This means that for a p-processor square machine on a 2-D mesh, $2p\sqrt{p}$ threads are needed to fill the machine, not p as one would hope for on a p-processor machine. This is the main reason that **Diff-1** performs poorly.

This problem is fixed in **Diff-2**, at the price of a small instability: the constants are set in such a way that a single thread can bounce back and forth between two processors on alternate diffusion steps. This same set of constants yields a "minimum slope" of 0, eliminating the requirement for $2p\sqrt{p}$ to fill the machine.

Unfortunately, even when there is no such interaction between integer queue lengths and stability requirements, the relaxation time is still $\Theta\left(l^2\right)$, where l is the diameter of the communications network [3]. This is very slow compared to the tree algorithms, which balance neighboring tree nodes in time proportional to the distance between the nodes, not the square of that distance.

Finally, the overhead of unneeded relaxation steps slows down all processors some fixed amount. This amount varies from one or two percent on four processors to more than 75 percent on a large machine with high t_n. This variation results from the fact that the cost of a diffusion step depends on the communication time between a processor and its nearest neighbors. For these reasons, even a perfectly balanced application achieves inferior performance under **Diff-1**and **Diff-2**.

Figures 7-10 through 7-13 lead us to the following conclusions:

1. The Round-Robin algorithms perform best of all real algorithms on machines of 256 or fewer processors, but their performance saturates early on larger machines.

2. **XTM** is competitive with simpler algorithms on small machines. On larger machines, **XTM** continues to achieve speedup, outperforming the Round-Robin thread managers by a large margin in some cases.

3. The Diffusion algorithms perform marginally worse than the others on small machines. As machine size is increased, **Diff-1** performs extremely poorly with respect to the others. **Diff-2** achieves somewhat better results than **Diff-1**, but is still significantly inferior to the others.

Again, as in the previous section, note that in Figure 7-13, the **P-Ideal** and **C-Ideal-1** results saturate very early in the region of interest. This is again due to a serialization inherent to those algorithms (see next section).

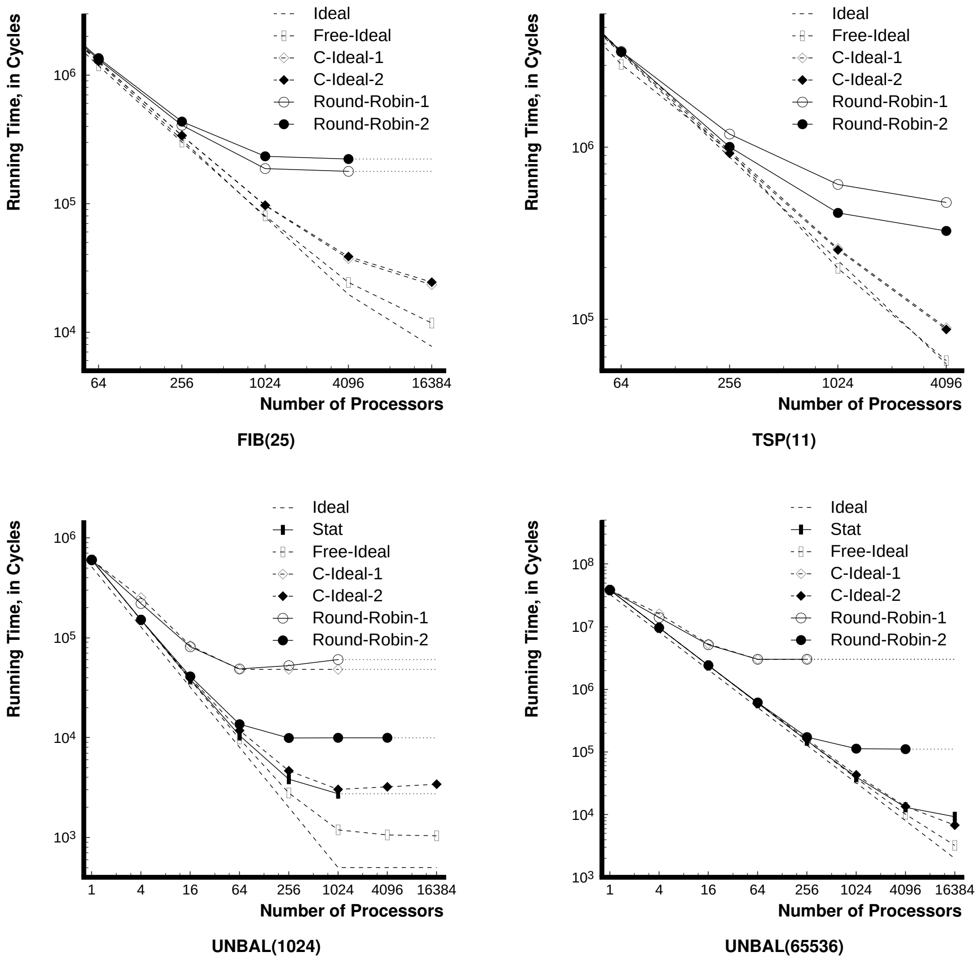


Figure 7-14: *Steal One vs. Steal Half.*

7.6 Steal-One vs. Steal-Half

When a processor from which threads can be stolen is located, consumer-based thread managers have a choice of policies when determining how much work to steal. The two choices we examine are *steal-one* and *steal-half*; they differentiate **C-Ideal-1** and **C-Ideal-2**, as well as **RR-1** and **RR-2**. For non-pathological applications, either choice can win out. However, when the initial load is severely unbalanced, as is the case with UNBAL, the performance of *steal-half* far exceeds that of *steal-one* (see Figure 7-14), due to serialization on the producer side.

For the UNBAL results shown, the poor performance curves for **C-Ideal-1** and **RR-1** are a direct result of the fact that only one thread at a time is taken from a processor whose queue contains runnable threads. Since UNBAL initially places all runnable threads on one processor, that processor becomes a bottleneck in the thread distribution process. It takes time for each thread-moving operation to run; if each thread is handled separately, then the rate at which threads can migrate from that processor to the rest of the machine is limited to the inverse of the amount of time it takes to move one thread off that processor. This "serialized" behavior puts a hard upper limit on the number of processors that can be effectively used.

Most applications do not display the kind of pathological behavior shown in UNBAL. All threads do not represent the same amount of work, and it is rare that a large number of small threads end up clustered in a very small region of a machine. However, UNBAL represents an extreme case of a machine whose load is severely unbalanced, and results of UNBAL runs give some insight into how a thread management scheme will behave in the presence of an unbalanced load.

For the other applications we simulated, the choice between *steal-one* and *steal-half* made very little difference. In the FIB graph in Figure 7-14, **RR-1** achieved slightly better results than **RR-2**; in the TSP graph, **RR-2** slightly outperformed **RR-1**. In both cases, there was no discernible difference in performance between **C-Ideal-1** and **C-Ideal-2**.

The tree-based algorithms are based on a policy that attempts to balance neighboring tree nodes evenly whenever one of the neighbors becomes empty. This section justifies this choice independently of the use of tree-type data structures. In particular, this section gives a case in which an uneven load balance will cause thread managers that employ a "steal-one" policy to exhibit serialized behavior. In other cases, the choice between *steal-half* and *steal-one* doesn't seem to be particularly important.

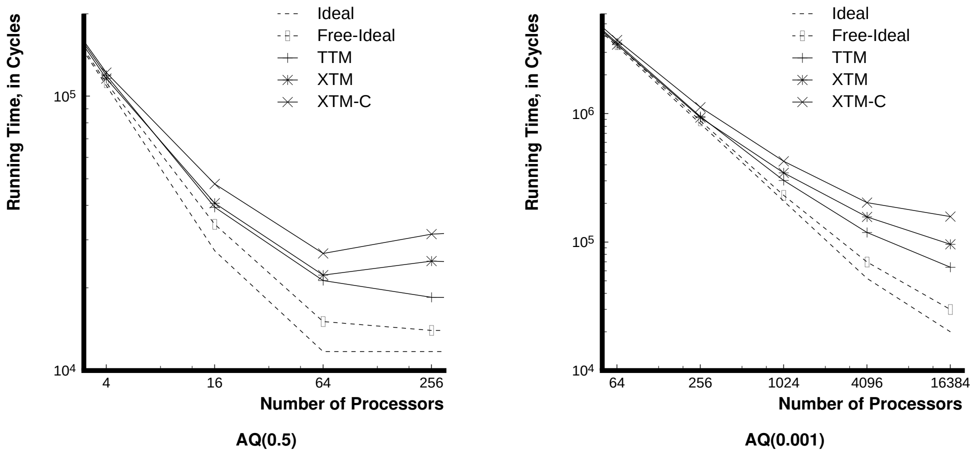


Figure 7-15: *AQ: Comparing Various Tree-Based Algorithms.*

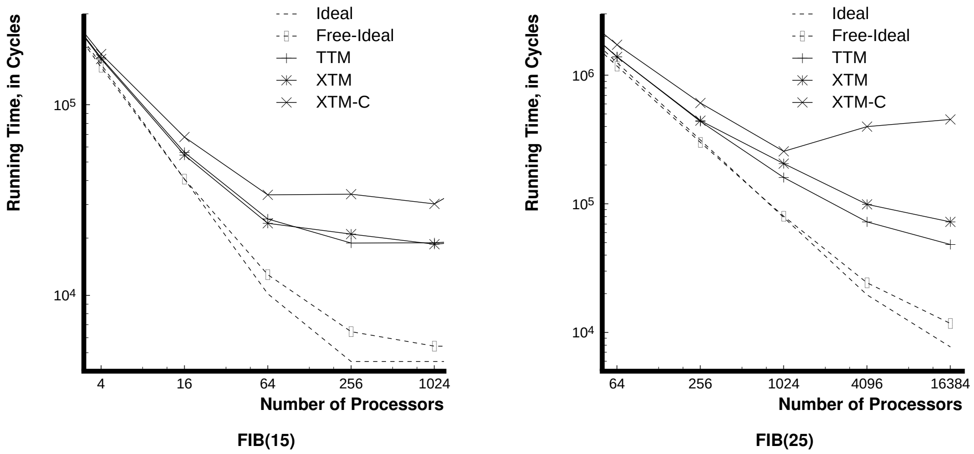


Figure 7-16: *FIB: Comparing Various Tree-Based Algorithms.*

7.7 Differences Between Tree-Based Algorithms

Three variants on a tree-based thread management theme were explored: **TTM**, **XTM** and **XTM-C**. **TTM** gives the best results for machines as large as the ones we measured, with $t_n = 1$. **XTM-C** performs very poorly on all but very slow networks for the following reasons:

1. The work estimates maintained at the tree nodes can be inaccurate, due to time delays inherent to the update process, inaccuracies built into the system to lower update costs, and, most importantly, the incorrect assumption that all threads are leaves in the application's task tree.

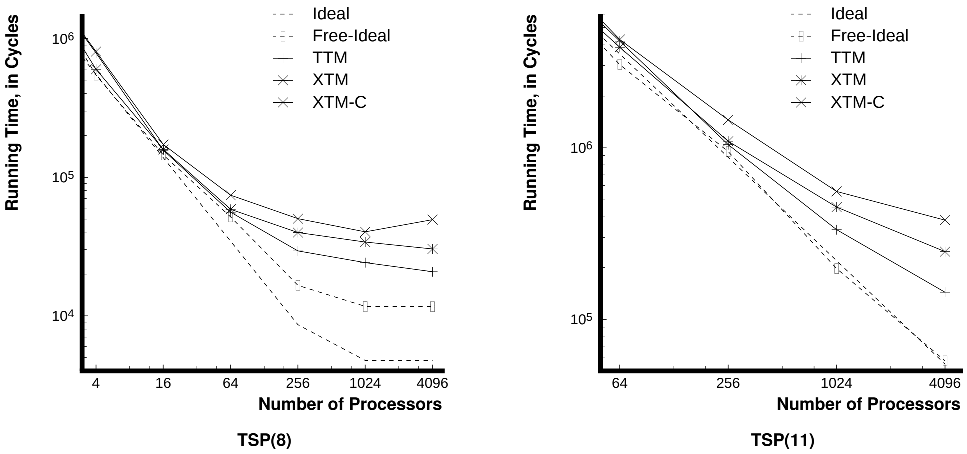


Figure 7-17: *TSP: Comparing Various Tree-Based Algorithms.*

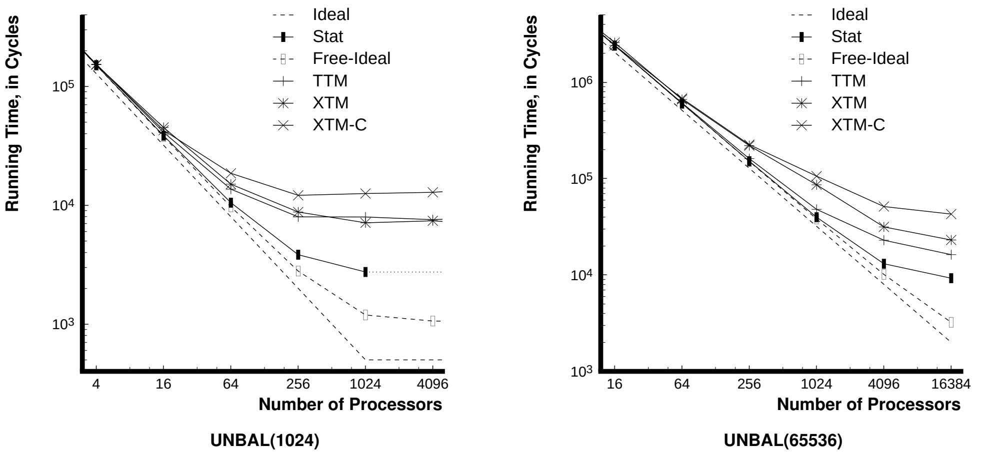


Figure 7-18: *UNBAL: – Comparing Various Tree-Based Algorithms.*

2. Maintaining work estimates in the tree carries significantly higher overhead than maintaining one-bit presence information. This added overhead results in correspondingly lower performance.

The data presented in Figures 7-15 through 7-18 suggests the following conclusions:

1. For the match between processor speed and network speed in Alewife (t_n= 1), **TTM** without the X-Tree's nearest-neighbor links is preferable, at least for machines containing up to 16,384 processors, which were the largest we could simulate. For a discussion of the effect of increasing t_n, see Section 7.8.

2. **XTM-C** never performs particularly well due to overhead costs and inaccurate weight estimates.

7.8 Faster Processors

Most of the data presented in this thesis assumes $t_n = 1$. Current technological trends suggest that in the future, the ratio between processor speed and network speed will become quite high. Since this research is primarily concerned with a world in which large-scale multiprocessors are commonplace, it is only natural that we should investigate a situation in which processors are very fast with respect to the interconnection network.

In addition, even PISCES simulations are orders of magnitude slower than actually running real programs on real hardware. This is compounded by the fact that simulating a p-processor multiprocessor on a uniprocessor is further slowed by a factor of p. When investigating the behavior of large machines, it would be very nice if simulations of smaller machines could in some way "fake" large-machine behavior. Artificially scaling the communication latency by increasing t_n seems to have just that effect, at least for the purposes of this thesis.

Figures 7-19 through 7-22 show the effect of increasing t_n from one cycle to 64 cycles. In each case, note that as the network slows down, the qualitative differences between the thread managers become more apparent on smaller machines. In particular, the drawbacks of **Diff-1**, **Diff-2**, **RR-2** and **XTM-C** show up more clearly. More interesting is the relation between **TTM** and **XTM**. As t_n is increased, the gains due to **XTM**'s near-neighbor links become more important, and the **XTM**'s performance surpasses that of **TTM**.

Another item of interest appears in Figure 7-20. Even for the case where $t_n = 1$, **XTM-C**'s performance takes a sharp dive on more than 1024 processors. For $t_n = 8$, the performance degradation begins at 256 processors, and for $t_n = 64$, performance was so bad that it was impractical to simulate. This poor performance results from the fact that **XTM-C** will not balance the load between two neighboring nodes unless the cost, which is measured as a function of the communication time between the two nodes, is outweighed by the advantage, which is predicted as a function of the amount of work the manager thinks is on the two nodes. For FIB in particular, these work estimates are poor, due to the system's lack of knowledge about how threads create other threads. It is therefore often the case that the **XTM-C** does not balance the workload on neighboring nodes when it should have done so. Note, however that when the manager knows about all the work in the system, as in Figure 7-22, **XTM-C**'s performance surpasses that of **XTM** for $t_n = 64$, by the slimmest of margins.

A final item of interest that occurs for large t_n concerns **Diff-1** and **Diff-2**. In all cases, the running time for the two Diffusion schedulers is actually longer on four processors than on one processor when $t_n = 64$. This is because on a machine with more than one processor, the cost of a diffusion step depends on t_n. When t_n is large, this overhead overwhelms the performance gains when going from one processor (no external communication takes place) to four processors (external communication takes place on every diffusion cycle).

The most important lesson to learn from the data presented in Figures 7-19 through 7-22 is the confirmation of asymptotic analysis. In all our analyses, we assumed that interprocessor communication is the dominating factor in large-scale system performance. When we adjust the ratio between computation and communication speeds so that this is the case for the machines we examined, the thread managers that yield good theoretical behaviors also yield good simulated behavior.

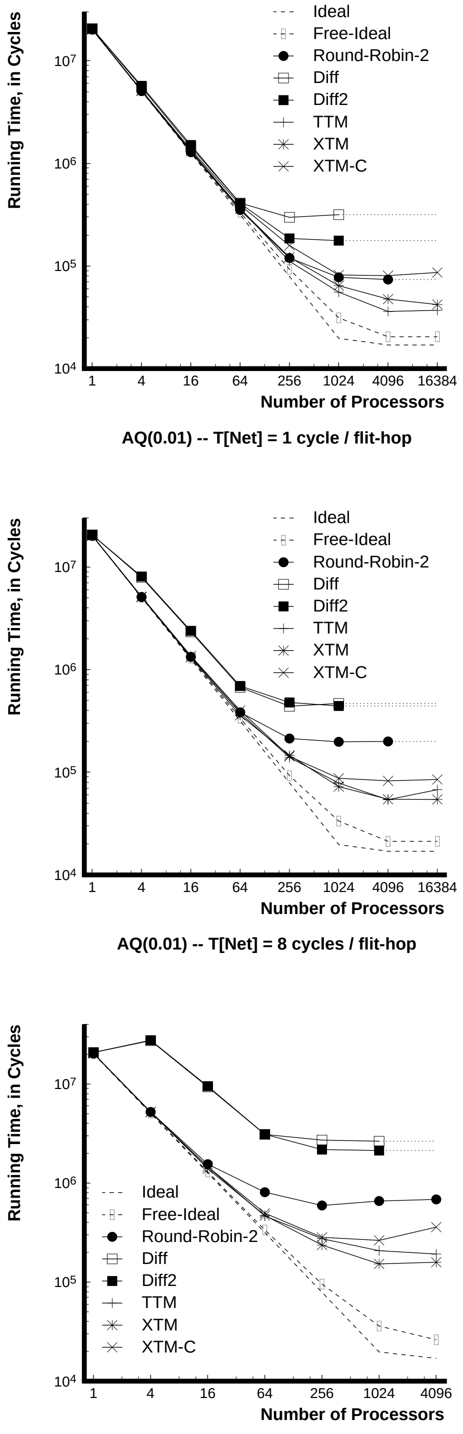


Figure 7-19: *AQ(0.01): Variable* t_n.

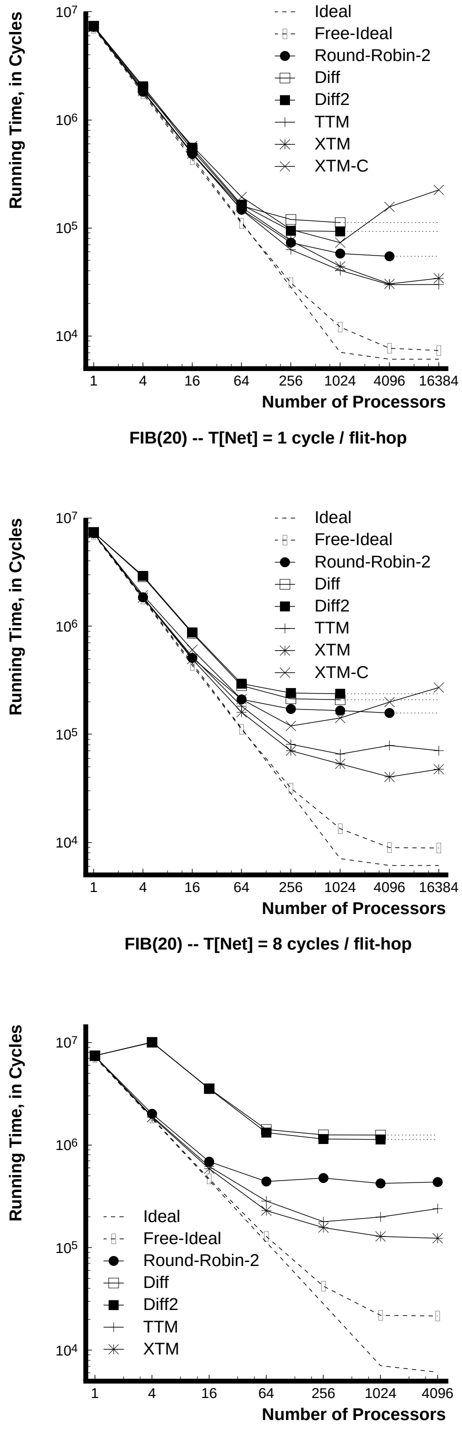


Figure 7-20: *FIB(20): Variable* t_n.

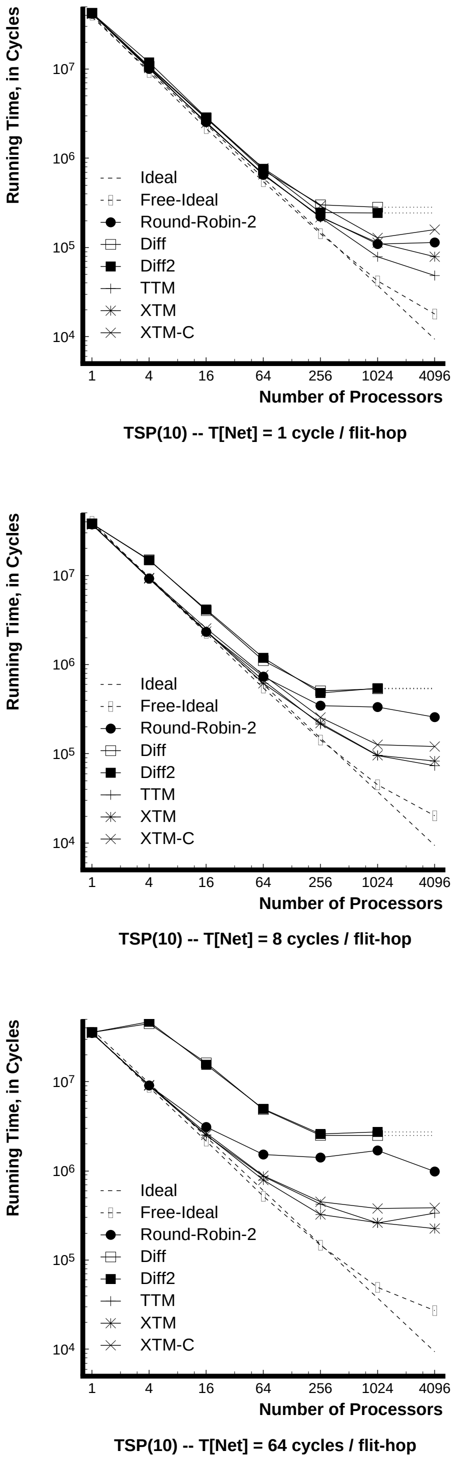


Figure 7-21: *TSP(10): Variable* t_n.

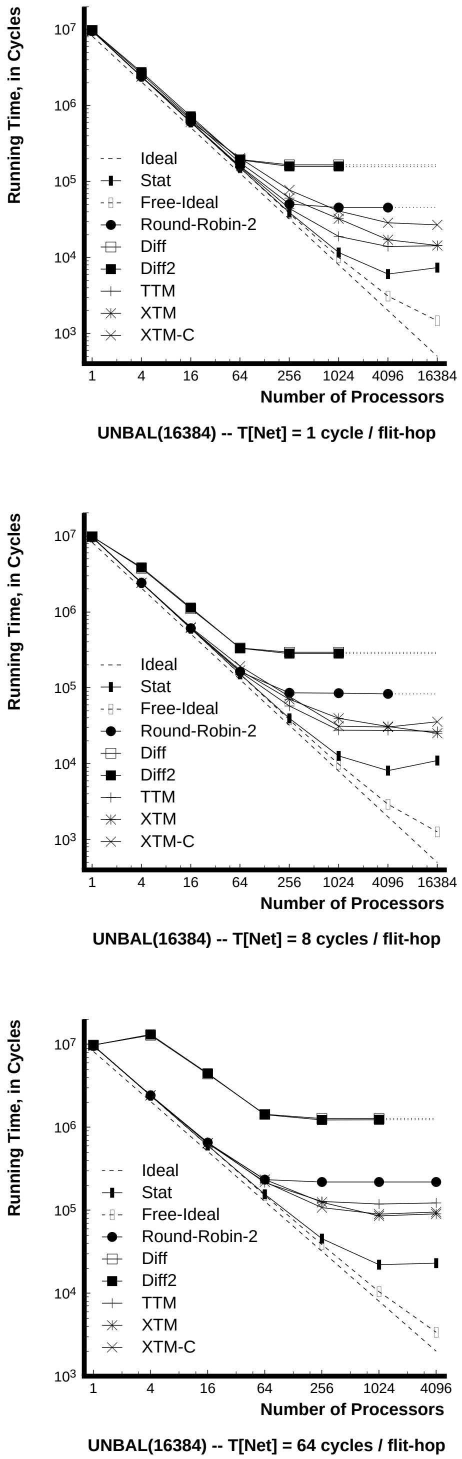


Figure 7-22: *UNBAL(16384): Variable* t_n.

7.9 MATMUL: The Effects of Static Locality

Finally, we look at MATMUL, a simple matrix multiply application. The two aspects of MATMUL performance we were most interested in were the effects of caching and the effects of different partitioning strategies. We therefore studied four cases: coarse-grained cached, fine-grained cached, coarse-grained uncached and fine-grained uncached. The data from these cases are presented in Figures 7-23 through 7-30.

When we say that MATMUL demonstrates strong static locality, we mean the following. In both the coarse-grained case and the fine-grained case, each thread accesses certain data elements in each matrix. The blocked algorithm breaks each matrix into a number of sub-blocks, which are spread out over the processor mesh in the obvious way. The overall multiply algorithm is then decomposed into sets smaller matrix multiplies, each of which finds the product of two sub-blocks (see Figures 6-8 and 6-9). Each thread accesses certain data elements of each matrix, in many cases more than once. A thread that accesses a block of data that resides on a given processor will run faster if it runs on or near to the processor on which its data is resident. Since the data is allocated in a static fashion, we say that the application demonstrates static locality.

Halstead and Ward[27] define *locality of reference* as follows:

> Reference to location X at time t implies that the probability of access to location $X + \Delta X$ at time $t + \Delta t$ increases as ΔX and Δt approach zero.

Caching is one mechanism that takes advantage of locality: when a remote data item is cached, multiple accesses to the item only pay the remote access cost once. Clearly, the behavior of an application that demonstrates locality will be strongly affected by the caching strategies of the machine on which the application is run. Since MATMUL is interesting primarily for its locality-related behavior, we decided to look at its running characteristics both in the presence and the absence of caches.

Except for **Stat**, all the thread management strategies we tested are dynamic, and can make use of no specific information about individual threads. Consequently, the managers have no way to make use of locality characteristics tying specific threads to specific processors, other than general heuristics that try to keep threads close to their point of origin.

For this reason, in the uncached case, all potential locality-related performance gains are inaccessible to the thread managers.

Two partitioning strategies for MATMUL are described in Chapter 6. The coarsely partitioned approach exhibits strong locality-related ties between each thread and the parts of the matrices it reads and writes. The finely partitioned approach loses much of that locality, but creates more threads, giving thread managers more flexibility. As might be expected, when caches are simulated, the coarsely-partitioned version runs much faster than the finely-partitioned version. For thread managers that exhibit load-sharing problems (diffusion algorithms, for example), the extra parallelism in the finely-partitioned version was necessary in order to avoid disastrous performance degradations.

Coarse Partitioning, with Caches

We now examine the details of the coarsely-partitioned version of MATMUL, with caches (see Figures 7-23 and 7-24). In this case, there is very little separation between the **Stat**, the idealized managers (**Free-Ideal**, **C-Ideal-1** and **C-Ideal-2**) and the tree-based managers (**TTM** and **XTM**). On large machines, the performance of the round-robin managers (**RR-1** and **RR-2**) begins to suffer. The diffusion managers (**Diff-1** and **Diff-2**) perform poorly for all problem sizes, machine sizes and network speeds.

Fine Partitioning, with Caches

For the finely-partitioned case with caches, the loss of locality due to the fine partitioning hurts the performance of all managers with respect to **Stat**(see Figures 7-25 and 7-26). The separations between managers observed for the coarsely-partitioned case nearly disappears, although the tree-based managers still perform marginally better than the other realizable managers. The managers' performance curves begin to separate out for large t_n, but the **Stat**always performs about twice as well as its nearest rival, primarily due to lower communication costs due to better locality behavior.

Coarse Partitioning, without Caches

Without caches, the system can't make use of the higher degree of locality in the coarsely-partitioned version (see Figures 7-27 and 7-28). However, the advantages of that locality are mostly lost to **Stat**, since local cache misses also carry a significant expense, so the results are similar in nature to the cached case, although the gap between **Stat** and the others is significantly larger than in the uncached case.

Fine Partitioning, without Caches

As in the finely-partitioned cached case, the finely-partitioned uncached case doesn't show much separation between the various thread managing strategies (see Figures 7-29 and 7-30). As expected, the diffusion algorithms perform poorly and for large machines, the round-robin managers do worse than the others.

MATMUL gives some insight into the behaviors of the various candidate thread managers when locality is an issue. Since a near-optimal static schedule can be derived from the regular structure of the application, the **Stat**always outperforms the other managers by a discernible margin. However, when a coarse partitioning strategy is used and when caches are available to recapture most of the locality inherent to the application, **TTM** and **XTM** perform nearly as well as **Stat**, as do **Free-Ideal**, **C-Ideal-1** and **C-Ideal-2**.

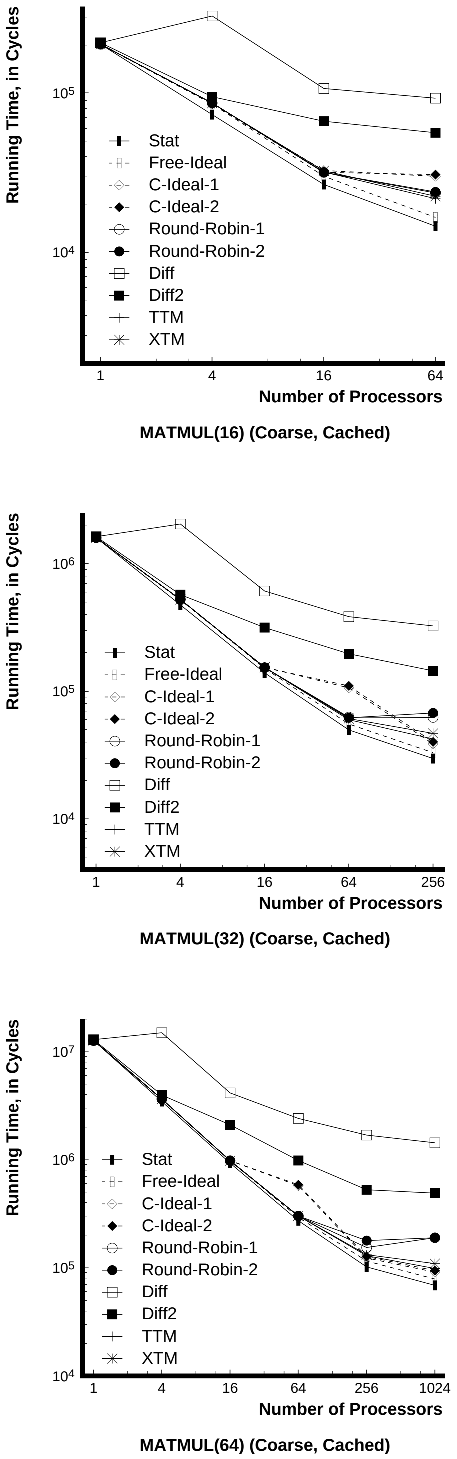


Figure 7-23: *MATMUL (Coarse, Cached).*

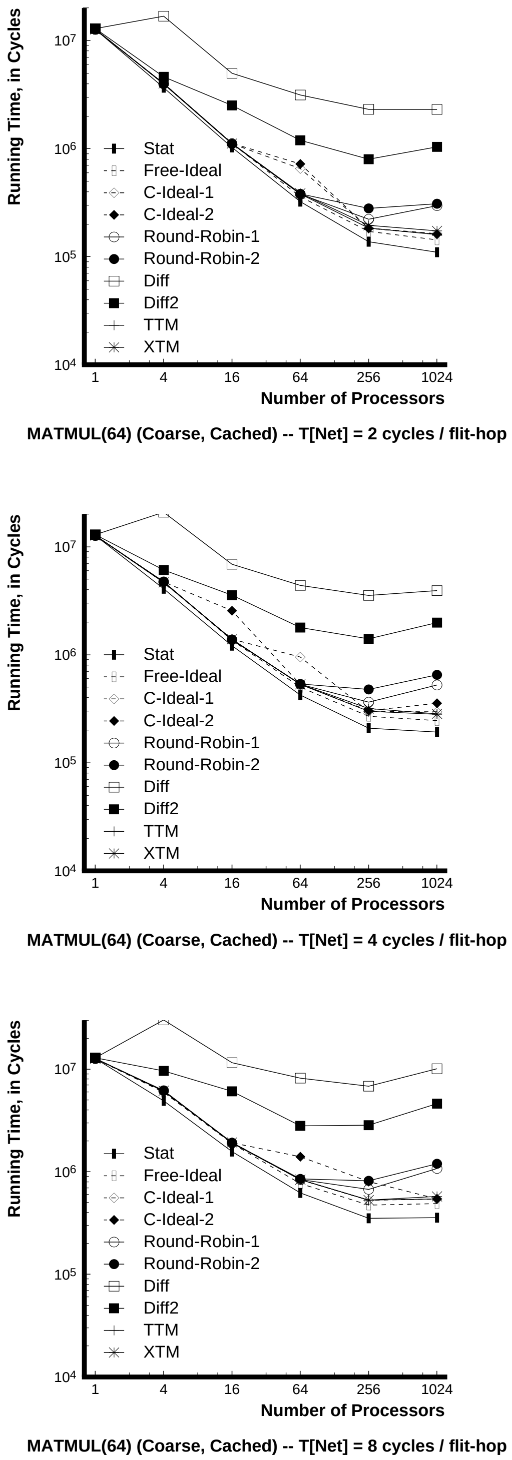


Figure 7-24: *MATMUL(64) (Coarse, Cached): Variable t_n.*

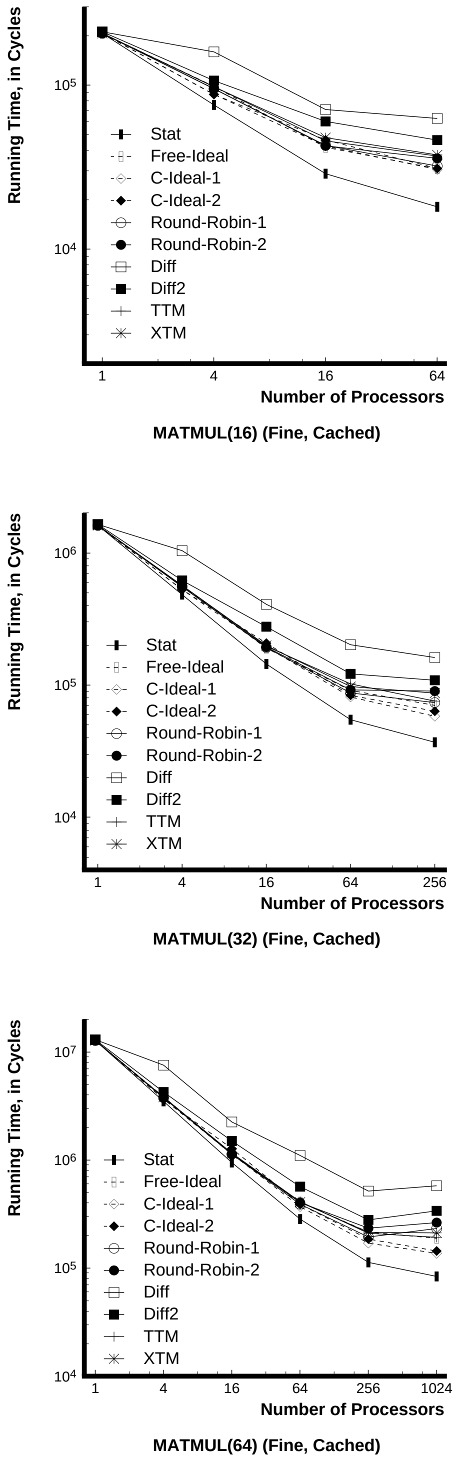


Figure 7-25: *MATMUL (Fine, Cached).*

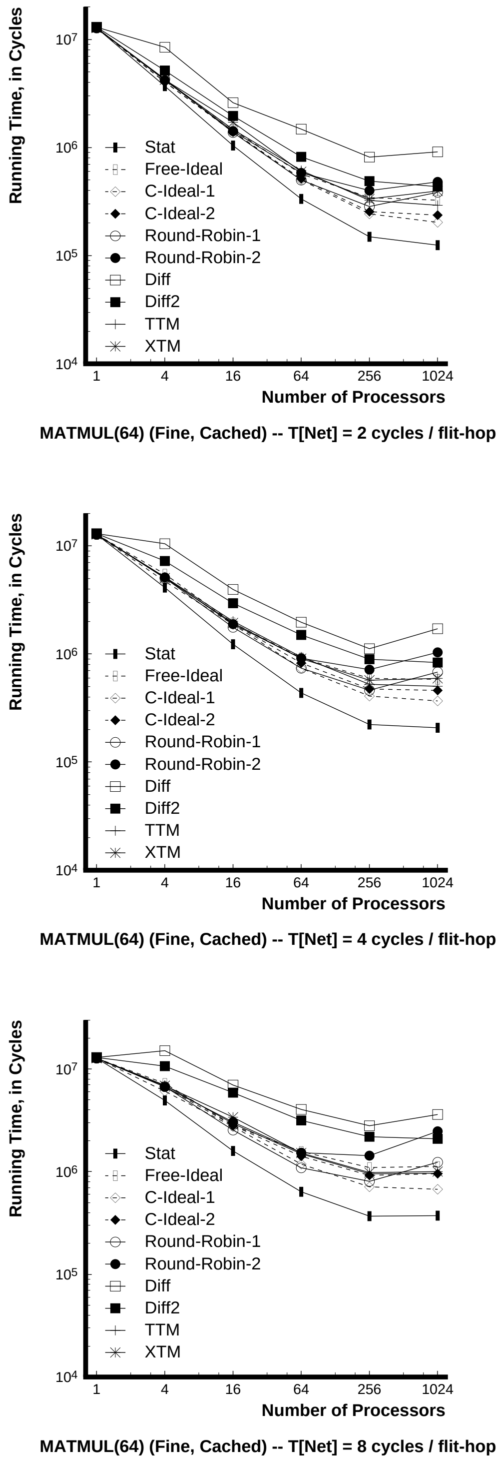
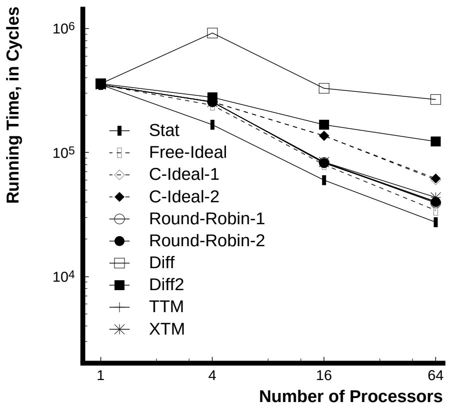
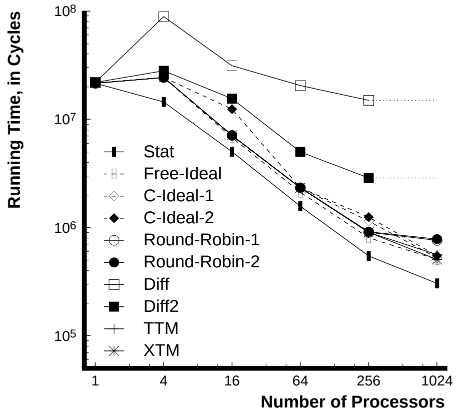


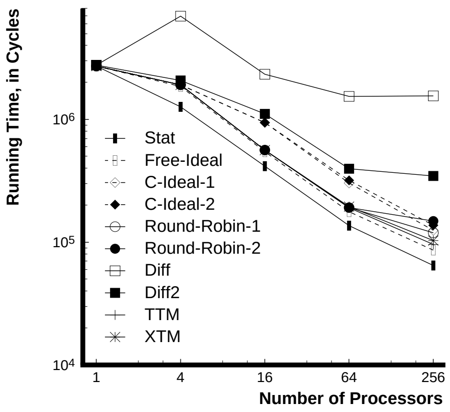
Figure 7-26: *MATMUL(64) (Fine, Cached): Variable t_n.*



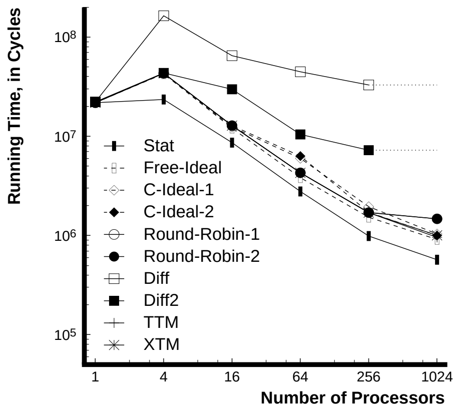
MATMUL(16) (Coarse, Uncached)



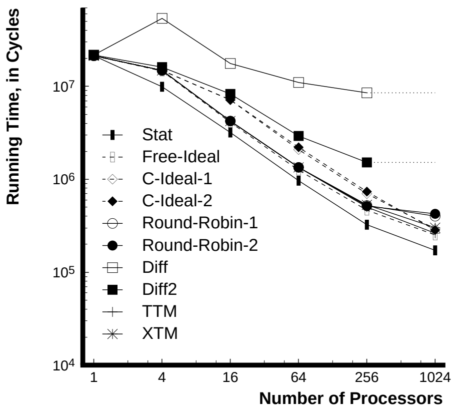
MATMUL(64) (Coarse, Uncached) -- T[Net] = 2 cycles / flit-hop



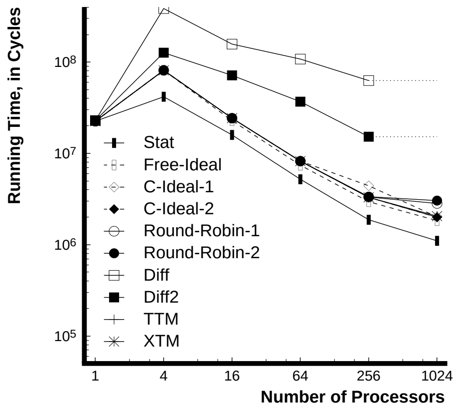
MATMUL(32) (Coarse, Uncached)



MATMUL(64) (Coarse, Uncached) -- T[Net] = 4 cycles / flit-hop



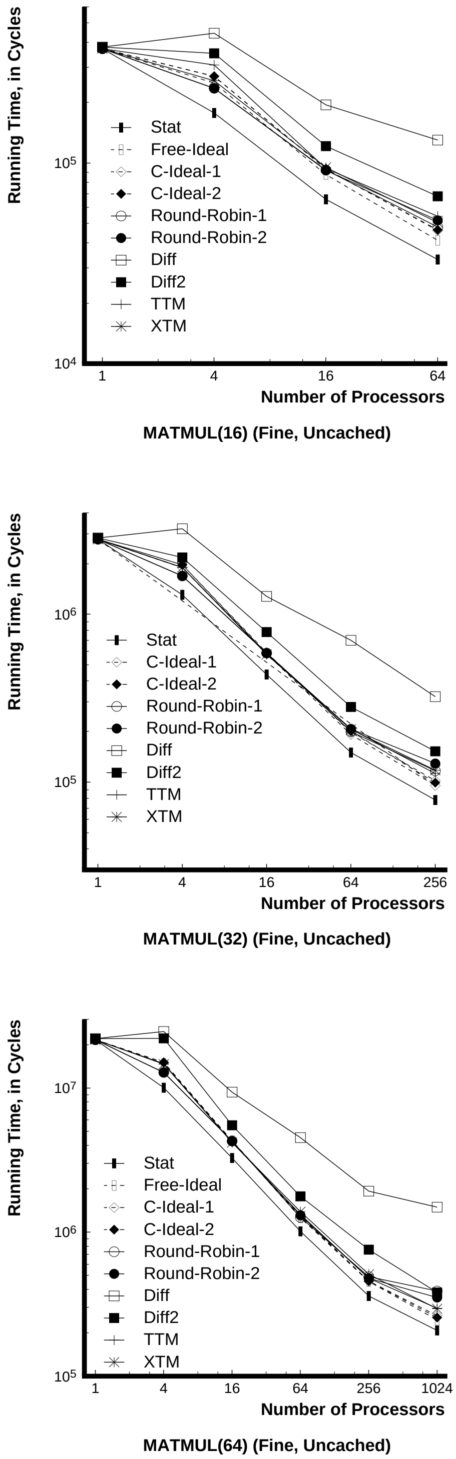
MATMUL(64) (Coarse, Uncached)



MATMUL(64) (Coarse, Uncached) -- T[Net] = 8 cycles / flit-hop

Figure 7-27: *MATMUL (Coarse, Uncached).*

Figure 7-28: *MATMUL(64) (Coarse, Uncached): Variable t_n.*



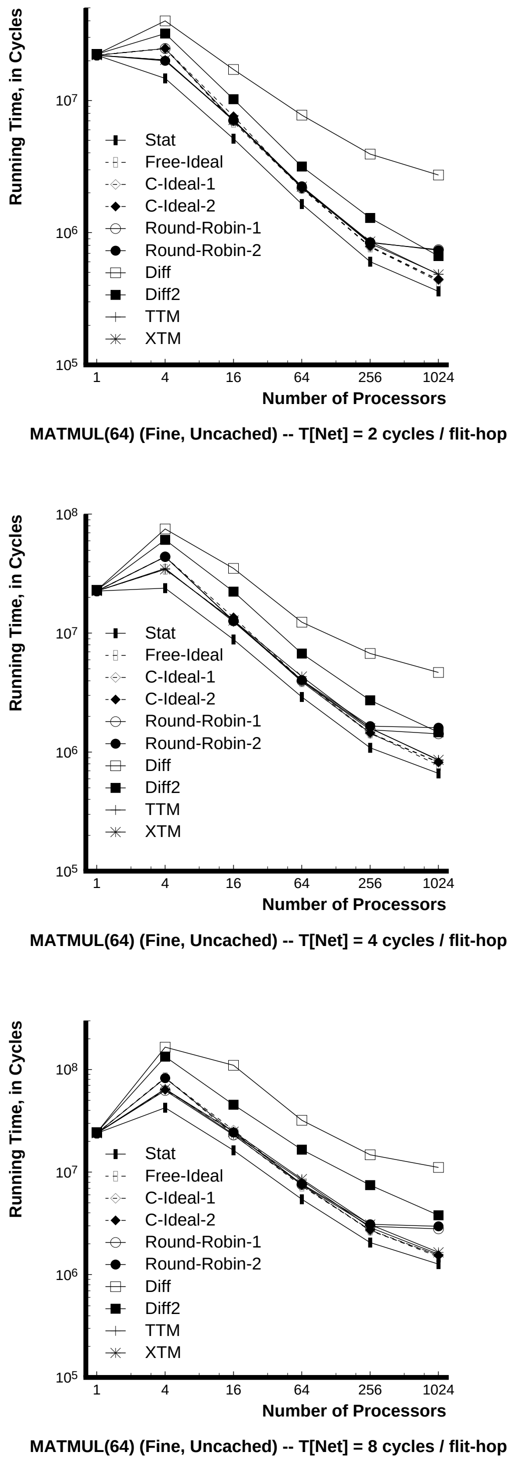


Figure 7-29: *MATMUL (Fine, Uncached).*

Figure 7-30: *MATMUL(64) (Fine, Uncached): Variable* t_n*.*

Chapter 8

Conclusions and Future Work

In the process of studying the behavior of **XTM** and other thread-management algorithms, we have learned a number of lessons, by both analytical and empirical means. Most importantly, we found that on small machines, there is no need to do anything clever about thread management: almost anything that avoids hot-spot behavior and that doesn't impose a high overhead will perform well. The results in Figures 7-2 through 7-5 point this out most clearly: for machines smaller than those in the "regions of interest," there is very little difference between **XTM**, one of the best realizable algorithms we tested, and **Diff-1**, one of the worst. The same figures show that on small machines, the added complexity of the tree-based algorithms doesn't cost very much; the tree-based thread managers work nearly as well as any of the others even where their added sophistication is not needed.

We also found that on large machines, communication locality becomes very important. One way to achieve lower communication costs is to use a message-passing style of computation, which is possible for well-understood statically structured algorithms. Chapter 5 give asymptotic cost arguments in favor of **XTM** to this effect. Second, the MATMUL results given in Chapter 7 show that when locality can be exploited to lower communication costs, it can lead to better program behavior. The fact that **Stat** runs were always the fastest points this out; the coarsely-partitioned case with caching also gives evidence to this effect.

Section 7.6 demonstrated that that parallel algorithms for large machines must avoid hot-spot behavior, or else risk losing the benefits of large-scale parallelism. Therefore, the tree-based thread-management algorithms presented in this thesis are all fully distributed, employing *combining* techniques for the collection and distribution of the threads being managed and of the global information needed to achieve good thread management. Other candidate thread-management algorithms that contained serialized behavior suffered severely when that serialized behavior became important (see Figure 7-14, for example).

We also learned that thread management is a global optimization problem. A good thread manager must therefore achieve efficient collection and distribution of relevant global information. One of the reasons that **TTM** and **XTM** work as well as they do is that they collect and distribute enough global information in order to make well-informed thread management decisions, without paying an excessive price for that information. **Diff-1**, **Diff-2**, **RR-1** and **RR-2** do not make use of global information and their performance suffers accordingly.

Although our analytical results point out the need for locality to minimize communication costs on large machines, it seems that for the particular set of parameters used for most of our simulations, the added locality gained by **XTM**'s near-neighbor links doesn't pay for the higher costs associated with passing presence information over those links, as compared with **TTM**. However, as as processor speeds increase with respect to network speeds, locality becomes more important and **XTM** performance surpasses that of **TTM**, as shown in Figures 7-19 through 7-22. More generally, as processor speed increases with respect to communication speed, "theoretically sound" thread management methodologies become necessary, even on smaller machines.

8.1 Future Work

There are a number of issues relevant to this research that we have left unresolved in this thesis. The first involves optimality proofs for the tree-based thread managers. In this thesis, our approach was to verify the good behavior of the overall algorithm through simulation. From the outset, we assumed that although we could analyze pieces of thread management algorithms with suitable approximations, when the pieces were assembled into an entire system, only simulations could give us insight into their behaviors. We have reason to believe, however, that we can make stronger formal statements about the behavior of our algorithms. It seems that the tree algorithms as a whole might be provably polylogcompetitive with the optimal: we intend to continue work in that area.

It would be useful to study how inaccuracy in information kept in the tree affects our analytical results. In general, the information kept in the tree is out-of-date: it takes time to migrate up from the leaves to higher-level nodes. In all our analyses, we assumed the information in the tree was accurate; it would be interesting to explore the effects of the potential inaccuracies. Along the same lines, we assumed, for analytical purposes, that all interprocessor messages were the same length. In fact, messages that carry more information (*e.g.*, a number of threads to be moved from one area of the machine to another) are different lengths. It would be useful to see what the varying of message lengths does to our results.

In Chapter 5, we give bounds on the cost of the update algorithms. These bounds are in some sense worst-case and best-case. The worst-case cost assumes there are very few threads on queues in the machine; the best-case cost is an expected cost given a sufficiently high, balanced workload. It would be useful to explore the region between these two extremes. For example, what kind of behavior is achieved if only a section of the machine has a sufficiently high workload or if we relax the balance criterion to some degree?

It would be nice to verify our simulation results on actual machines. Perhaps the 128-processor CM-5 recently purchased by the lab would be useful for that purpose, or maybe a 128-processor Alewife machine that is planned to be built sometime next year. Although by our predictions, in many cases, 128 processors is not large enough to be interesting, perhaps we can observe some of the predicted trends beginning to occur. Furthermore, if we can figure out how to artificially increase t_n, 128 processors may be big enough to encounter large-machine behavior.

It would be interesting to pay more attention to the effects of locality inherent to the applications being managed. Although the algorithms that make up **XTM** have good locality, the applications being managed are kept local inasmuch as **XTM** tries to keep threads close to their point of origin; other than that, no attention is paid to locality in the application. It would be interesting to study how well this heuristic works on an application that carries a higher degree of inter-thread communication.

Finally, this work assumed no communication path from the compiler to the runtime system. In some cases, the compiler should be able to distill information about the running characteristics of a given application that the runtime system can use. What form would that information take, such that the compiler can extract it and the runtime system can put it to work? Perhaps some work on compiler-generated annotations of individual threads or profile-driven compilation would be appropriate for this purpose.

Appendix A

Presence Bit Update Algorithm

The following pseudocode gives a formal statement of the presence bit update algorithm.

```
Update_Presence_Bit(Node)
{
    if (   (Node_Presence_Bit(Node) == 0)
        && (   (   Node_Is_Leaf_Node(Node)
                && Thread_Queue_Not_Empty(Node))
            || (   Node_Not_Leaf_Node(Node)
                && Child_Presence_Bits_Not_All_Zero(Node))))

        Node_Presence_Bit(Node) = 1;
        Update_Neighbor_Presence_Bit_Caches(Node, 1);
        if (Node_Has_Parent(Node))
            Update_Parent_Presence_Bit_Cache(Node, 1);
            Update_Presence_Bit(Node_Parent(Node));

    else if (   (Node_Presence_Bit(Node) == 1)
             && (   (   Node_Is_Leaf_Node(Node)
                     && Thread_Queue_Is_Empty(Node))
                 || (   Node_Not_Leaf_Node(Node)
                     && Child_Presence_Bits_Are_All_Zero(Node))))

        Node_Presence_Bit(Node) = 0;
        Update_Neighbor_Presence_Bit_Caches(Node, 0);
        if (Node_Has_Parent(Node))
            Update_Parent_Presence_Bit_Cache(Node, 0);
            Update_Presence_Bit(Node_Parent(Node));
}
```

Appendix B

Application Code

B.1 AQ

```
;; See pgms/aq/faq.c for documentation.  Adapted from SemiC output.

(herald FAQ)

(DEFINE (MAIN LEV)
  (set *task-cycles* 1740)
  (LET ((TOL (CASE LEV
               ((8) 0.5)
               ((7) 0.1)
               ((6) 0.05)
               ((5) 0.01)
               ((4) 0.005)
               ((3) 0.001)
               ((2) 5.0E-4)
               ((1) 1.0E-4)
               (ELSE 5.0E-5))))
    (AQ 0.0 0.0 2.0 2.0 TOL (Q 0.0 0.0 2.0 2.0))))


(DEFINE (Q X0 Y0 X1 Y1)
  (LET ((DX (- X1 X0))
        (DY (- Y1 Y0))
        (XM (/ (+ X0 X1) 2.0))
        (YM  (/ (+ Y0 Y1) 2.0)))
    (/ (* (+ (F X0 Y0) (F X0 Y1) (F X1 Y0) (F X1 Y1)
             (* 2.0 (F XM YM)))
          DX DY)
       6.0)))


(DEFINE (F X Y)
  (LET* ((R0 (* X Y))
         (R1 (* R0 R0)))
    (* R1 R1)))
```

```
(DEFINE (AQ X0 Y0 X1 Y1 TOLERANCE Q0)
  (LET* ((XM (/ (+ X0 X1) 2.0))
         (YM (/ (+ Y0 Y1) 2.0))
         (Q1 (Q X0 Y0 XM YM))
         (Q2 (Q XM Y0 X1 YM))
         (Q3 (Q X0 YM XM Y1))
         (Q4 (Q XM YM X1 Y1))
         (SUM (+ Q1 Q2 Q3 Q4)))
    (IF (< (IF (> SUM Q0) (- SUM Q0) (- Q0 SUM)) TOLERANCE)
        (BLOCK (PAUSE 1000) SUM)
        (LET* ((TOLERANCE (/ TOLERANCE 4.0))
               (SUM0 (BLOCK (PAUSE 1220)
                            (FUTURE (AQ X0 Y0 XM YM TOLERANCE Q1))))
               (SUM1 (BLOCK (PAUSE 80)
                            (FUTURE (AQ XM Y0 X1 YM TOLERANCE Q2))))
               (SUM2 (BLOCK (PAUSE 80)
                            (FUTURE (AQ X0 YM XM Y1 TOLERANCE Q3))))
               (SUM3 (BLOCK (PAUSE 80)
                            (FUTURE (AQ XM YM X1 Y1 TOLERANCE Q4))))
               (VAL (BLOCK (PAUSE 100)
                           (+ (TOUCH SUM0) (TOUCH SUM1)
                              (TOUCH SUM2) (TOUCH SUM3)))))
          (PAUSE 100)
          VAL))))
```

B.2 FIB

```
(herald ffib)

(define (ffib n)
  (if (fx<= n 2)
      (block
        (pause 60)
        1)
      (let* ((lhs (block
                    (pause 62)
                    (future (ffib (fx- n 1)))))
             (rhs (block
                    (pause 41)
                    (future (ffib (fx- n 2)))))
             (val (block
                    (pause 229)
                    (fx+ (touch lhs) (touch rhs)))))
        (pause 66)
        val)))

(define (main x)
  (set *task-cycles* 398)
  (ffib x))
```

B.3 TSP

```
;;;
;;; Branch-and-bound TSP solver.
;;;
;;; Objective: find shortest tour of all N cities starting at city 0.
;;;

(herald tsp)

(define (main n ordered?)
  (set *task-cycles* (* 300 n))
  (if (fx< n 1)
      (error "TSP solver only works with 1 or more cities!"))
  (let ((initial-path (make-path n))
        (first-guess (make-path n)))
    (dotimes (i n) (set (path-element first-guess i) i))
    (set (path-element initial-path 0) 0)
    (let ((ic-d-mat (cond ((null? ordered?)
                           (vref unordered-ic-d-mats n))
                          ((eq? ordered? '#t)
                           (vref ordered-ic-d-mats n))
                          (else (vref opt-ordered-ic-d-mats n))))
          (cities (cond ((null? ordered?) (nth unordered-cities n))
                        ((eq? ordered? '#t) (nth ordered-cities n))
                        (else (nth opt-ordered-cities n)))))
      (init-best-so-far first-guess ic-d-mat cities)
      (let* ((s-path
               (find-shortest-path initial-path 1 n ic-d-mat cities))
             (s-path (if (null? s-path) first-guess s-path)))
        (message (format nil "Shortest Path: ~s"
                         (output-path s-path cities)))
        (message (format nil "Shortest Len : ~d"
                         (path-length s-path ic-d-mat)))
        s-path))))
```

```
;;;
;;; main recursive path finder: returns #f if no path shorter than
;;;  current best along current path.
;;;

(define (find-shortest-path path-so-far step-index n-cities ic-d-mat
         cities)
  (cond ((fx>= (path-length path-so-far ic-d-mat) (best-so-far))
         (pause 50)
         '#f)
        ((fx>= step-index n-cities)
         (message (format nil "New Best Path: ~s"
                          (output-path path-so-far cities)))
         (message (format nil "New Best Len : ~s"
                          (path-length path-so-far ic-d-mat)))
         (set (best-so-far) (path-length path-so-far ic-d-mat))
         (pause 50)
         path-so-far)
        (else
         (pause 50)
         (iterate loop ((paths '()) (next-city 0))
           (cond ((fx>= next-city n-cities)
                  (pause 50)
                  (select-best-path paths ic-d-mat))
                 ((city-part-of-path next-city path-so-far)
                  (pause 50)
                  (loop paths (fx+ next-city 1)))
                 (else
                  (let ((new-path (copy-path path-so-far)))
                    (set (path-element new-path step-index) next-city)
                    (pause 200)
                    (loop (cons
                           (future
                            (find-shortest-path new-path
                                                (fx+ step-index 1)
                                                n-cities
                                                ic-d-mat
                                                cities))
                               paths)
                          (fx+ next-city 1)))))))))
```

```
(define (select-best-path paths ic-d-mat)
  (iterate loop ((best-path '#f) (best-path-length '#f) (paths paths))
    (if (null? paths)
        best-path
        (let ((candidate (touch (car paths))))
          (cond ((null? candidate)
                 (pause 50)
                 (loop best-path best-path-length (cdr paths)))
                ((null? best-path-length)
                 (pause 50)
                 (loop candidate
                      (path-length candidate ic-d-mat)
                      (cdr paths)))
                (else
                 (let ((current-length
                        (path-length candidate ic-d-mat)))
                   (pause 100)
                   (if (fx< current-length best-path-length)
                       (loop candidate current-length (cdr paths))
                       (loop best-path best-path-length (cdr paths))))
                 ))))))
```

```
;;;
;;; Best-so-far abstraction
;;;

(lset *best-so-far* '#f)

(define (init-best-so-far first-guess ic-d-mat cities)
  (let ((len (path-length first-guess ic-d-mat)))
    (message (format nil "Initial Best Path: ~s"
                     (output-path first-guess cities)))
    (message (format nil "Initial Best Len : ~s" len))
    (set *best-so-far* (make-vector *N-Processors* len))))

(define-constant best-so-far
  (object (lambda ()
            (vref *best-so-far* *my-pid*))
    ((setter self)
     (lambda (len)
       (broadcast (row col)
         (when (fx< len (best-so-far))
           (set (vref *best-so-far* *my-pid*) len)))))))
```

```
;;;
;;; PATH abstraction
;;;

(define-constant path-element
  (object (lambda (path elt)
            (vref path elt))
    ((setter self)
     (lambda (path elt val)
       (set (vref path elt) val)))))

(define (make-path n)
  (make-vector n '#f))

(define copy-path copy-vector)

(define (city-part-of-path city path)
  (let ((len (path-steps path)))
    (iterate loop ((i 0))
      (cond ((fx>= i len) '#f)
            ((eq? city (path-element path i)) '#t)
            (else (loop (fx+ i 1)))))))

(define (path-length path ic-d-mat)
  (let ((len (path-steps path)))
    (iterate loop ((step 1) (p-city (path-element path 0)) (sum 0))
      (if (fx>= step len)
          sum
          (let ((current-city (path-element path step)))
            (if (null? current-city)
                sum
                (loop
                 (fx+ step 1)
                 current-city
                 (fx+ sum
                 (ic-dist p-city current-city ic-d-mat))))))))))

(define-integrable (path-steps path)
  (vector-length path))

(define (output-path path cities)
  (cons path
        (iterate loop ((i 0) (coords '()))
          (if (fx>= i (path-steps path))
              (reverse coords)
              (loop (fx+ i 1)
                    (cons (nth cities (path-element path i))
                          coords))))))

(define-constant (ic-dist x y ic-mat)
  (vref (vref ic-mat x) y))
```

```
(herald cities)

;;;
;;; Lists of city coordinates (unordered).
;;;

(define unordered-cities
  '(()
    ((-1 . 1))
    ((-2 . 0) (-2 . -1))
    ((-1 . 3) (3 . -1) (-1 . 0))
    ((-3 . -1) (-2 . 1) (0 . 2) (-1 . 4))
    ((-5 . -3) (-2 . -2) (-1 . 3) (1 . 2) (1 . 0))
    ((6 . 6) (6 . 1) (0 . 0) (2 . 0) (-4 . 1) (-6 . -6))
    ((-1 . 3) (-5 . 0) (1 . 5) (-2 . 1) (5 . -5) (-3 . 6) (3 . 6))
    ((4 . -1) (-6 . 3) (-1 . 4) (2 . -5) (7 . 7) (3 . -4) (8 . 2)
              (-1 . 6))
    ((1 . 7) (1 . 4) (-5 . 1) (6 . -6) (-9 . -2) (1 . 4) (8 . -6)
             (5 . -2) (-4 . 9))
    ((2 . -8) (1 . 0) (7 . -10) (-3 . -9) (-6 . 8) (-5 . -8)
              (2 . -9) (-3 . 3) (1 . 7) (10 . 10))
    ((5 . -6) (-1 . -9) (-2 . 0) (1 . -9) (8 . -9) (-2 . 6) (0 . 10)
              (5 . 1) (7 . 1) (-9 . 9) (0 . 6))
    ((11 . 11) (-9 . 12) (8 . -3) (12 . -10) (5 . -4) (-11 . 9)
               (-1 . 3) (-4 . 9) (-3 . -2) (-9 . 10) (-7 . 11)
               (-7 . 6))))

;;;
;;; Identical lists of city coordinates (ordered by greedy algorithm).
;;;

(define ordered-cities
  '(()
    ((-1 . 1))
    ((-2 . 0) (-2 . -1))
    ((-1 . 3) (-1 . 0) (3 . -1))
    ((-3 . -1) (-2 . 1) (0 . 2) (-1 . 4))
    ((-5 . -3) (-2 . -2) (1 . 0) (1 . 2) (-1 . 3))
    ((6 . 6) (6 . 1) (2 . 0) (0 . 0) (-4 . 1) (-6 . -6))
    ((-1 . 3) (1 . 5) (3 . 6) (-3 . 6) (-2 . 1) (-5 . 0) (5 . -5))
    ((4 . -1) (3 . -4) (2 . -5) (-1 . 4) (-1 . 6) (-6 . 3) (7 . 7)
              (8 . 2))
    ((1 . 7) (1 . 4) (1 . 4) (-5 . 1) (-9 . -2) (-4 . 9) (5 . -2)
             (6 . -6) (8 . -6))
    ((2 . -8) (2 . -9) (7 . -10) (-3 . -9) (-5 . -8) (1 . 0) (-3 . 3)
              (-6 . 8) (1 . 7) (10 . 10))
    ((5 . -6) (8 . -9) (1 . -9) (-1 . -9) (-2 . 0) (-2 . 6) (0 . 6)
              (0 . 10) (-9 . 9) (5 . 1) (7 . 1))
    ((11 . 11) (8 . -3) (5 . -4) (-3 . -2) (-1 . 3) (-4 . 9)
               (-7 . 11) (-9 . 12) (-9 . 10) (-11 . 9) (-7 . 6)
               (12 . -10))))
```

```
;;;
;;; Identical lists of city coordinates (optimal ordering).
;;;

(define opt-ordered-cities
  '(()
    ((-1 . 1))
    ((-2 . 0) (-2 . -1))
    ((-1 . 3) (-1 . 0) (3 . -1))
    ((-3 . -1) (-2 . 1) (0 . 2) (-1 . 4))
    ((-5 . -3) (-2 . -2) (1 . 0) (1 . 2) (-1 . 3))
    ((6 . 6) (6 . 1) (2 . 0) (0 . 0) (-4 . 1) (-6 . -6))
    ((-1 . 3) (1 . 5) (3 . 6) (-3 . 6) (-5 . 0) (-2 . 1) (5 . -5))
    ((4 . -1) (2 . -5) (3 . -4) (8 . 2) (7 . 7) (-1 . 4) (-1 . 6)
             (-6 . 3))
    ((1 . 7) (-4 . 9) (-9 . -2) (-5 . 1) (1 . 4) (1 . 4) (5 . -2)
            (6 . -6) (8 . -6))
    ((2 . -8) (7 . -10) (2 . -9) (-3 . -9) (-5 . -8) (1 . 0) (-3 . 3)
             (-6 . 8) (1 . 7) (10 . 10))
    ((5 . -6) (8 . -9) (1 . -9) (-1 . -9) (-2 . 0) (5 . 1) (7 . 1)
             (0 . 6) (-2 . 6) (0 . 10) (-9 . 9))
    ((11 . 11) (-4 . 9) (-7 . 11) (-9 . 12) (-9 . 10) (-11 . 9)
              (-7 . 6) (-1 . 3) (-3 . -2) (5 . -4) (8 . -3)
              (12 . -10))))
```

```
;;;
;;; Intercity distance matrices for unordered cities
;;;

(define unordered-ic-d-mats
  '#(#()

     #(#(0))

     #(#(0 1)
       #(1 0))

     #(#(0 5 3)
       #(5 0 4)
       #(3 4 0))

     #(#(0 2 4 5)
       #(2 0 2 3)
       #(4 2 0 2)
       #(5 3 2 0))

     #(#(0 3 7 7 6)
       #(3 0 5 5 3)
       #(7 5 0 2 3)
       #(7 5 2 0 2)
       #(6 3 3 2 0))

     #(#( 0  5  8  7 11 16)
       #( 5  0  6  4 10 13)
       #( 8  6  0  2  4  8)
       #( 7  4  2  0  6 10)
       #(11 10  4  6  0  7)
       #(16 13  8 10  7  0))

     #(#( 0  5  2  2 10  3  5)
       #( 5  0  7  3 11  6 10)
       #( 2  7  0  5 10  4  2)
       #( 2  3  5  0  9  5  7)
       #(10 11 10  9  0 13 11)
       #( 3  6  4  5 13  0  6)
       #( 5 10  2  7 11  6  0))

     #(#( 0 10  7  4  8  3  5  8)
       #(10  0  5 11 13 11 14  5)
       #( 7  5  0  9  8  8  9  2)
       #( 4 11  9  0 13  1  9 11)
       #( 8 13  8 13  0 11  5  8)
       #( 3 11  8  1 11  0  7 10)
       #( 5 14  9  9  5  7  0  9)
       #( 8  5  2 11  8 10  9  0))
```

```
#(#( 0  3  8 13 13  3 14  9  5)
  #( 3  0  6 11 11  0 12  7  7)
  #( 8  6  0 13  5  6 14 10  8)
  #(13 11 13  0 15 11  2  4 18)
  #(13 11  5 15  0 11 17 14 12)
  #( 3  0  6 11 11  0 12  7  7)
  #(14 12 14  2 17 12  0  5 19)
  #( 9  7 10  4 14  7  5  0 14)
  #( 5  7  8 18 12  7 19 14  0))

#(#( 0  8  5  5 17  7  1 12 15 19)
  #( 8  0 11  9 10 10  9  5  7 13)
  #( 5 11  0 10 22 12  5 16 18 20)
  #( 5  9 10  0 17  2  5 12 16 23)
  #(17 10 22 17  0 16 18  5  7 16)
  #( 7 10 12  2 16  0  7 11 16 23)
  #( 1  9  5  5 18  7  0 13 16 20)
  #(12  5 16 12  5 11 13  0  5 14)
  #(15  7 18 16  7 16 16  5  0  9)
  #(19 13 20 23 16 23 20 14  9  0))

#(#( 0  6  9  5  4 13 16  7  7 20 13)
  #( 6  0  9  2  9 15 19 11 12 19 15)
  #( 9  9  0  9 13  6 10  7  9 11  6)
  #( 5  2  9  0  7 15 19 10 11 20 15)
  #( 4  9 13  7  0 18 20 10 10 24 17)
  #(13 15  6 15 18  0  4  8 10  7  2)
  #(16 19 10 19 20  4  0 10 11  9  4)
  #( 7 11  7 10 10  8 10  0  2 16  7)
  #( 7 12  9 11 10 10 11  2  0 17  8)
  #(20 19 11 20 24  7  9 16 17  0  9)
  #(13 15  6 15 17  2  4  7  8  9  0))

#(#( 0 20 14 21 16 22 14 15 19 20 18 18)
  #(20  0 22 30 21  3 12  5 15  2  2  6)
  #(14 22  0  8  3 22 10 16 11 21 20 17)
  #(21 30  8  0  9 29 18 24 17 29 28 24)
  #(16 21  3  9  0 20  9 15  8 19 19 15)
  #(22  3 22 29 20  0 11  7 13  2  4  5)
  #(14 12 10 18  9 11  0  6  5 10 10  6)
  #(15  5 16 24 15  7  6  0 11  5  3  4)
  #(19 15 11 17  8 13  5 11  0 13 13  8)
  #(20  2 21 29 19  2 10  5 13  0  2  4)
  #(18  2 20 28 19  4 10  3 13  2  0  5)
  #(18  6 17 24 15  5  6  4  8  4  5  0))))
```

```
;;;
;;; Intercity distance matrices for ordered cities
;;;

(define ordered-ic-d-mats
  '#(#()

     #(#(0))

     #(#(0 1)
       #(1 0))

     #(#(0 3 5)
       #(3 0 4)
       #(5 4 0))

     #(#(0 2 4 5)
       #(2 0 2 3)
       #(4 2 0 2)
       #(5 3 2 0))

     #(#(0 3 6 7 7)
       #(3 0 3 5 5)
       #(6 3 0 2 3)
       #(7 5 2 0 2)
       #(7 5 3 2 0))

     #(#( 0  5  7  8 11 16)
       #( 5  0  4  6 10 13)
       #( 7  4  0  2  6 10)
       #( 8  6  2  0  4  8)
       #(11 10  6  4  0  7)
       #(16 13 10  8  7  0))

     #(#( 0  2  5  3  2  5 10)
       #( 2  0  2  4  5  7 10)
       #( 5  2  0  6  7 10 11)
       #( 3  4  6  0  5  6 13)
       #( 2  5  7  5  0  3  9)
       #( 5  7 10  6  3  0 11)
       #(10 10 11 13  9 11  0))

     #(#( 0  3  4  7  8 10  8  5)
       #( 3  0  1  8 10 11 11  7)
       #( 4  1  0  9 11 11 13  9)
       #( 7  8  9  0  2  5  8  9)
       #( 8 10 11  2  0  5  8  9)
       #(10 11 11  5  5  0 13 14)
       #( 8 11 13  8  8 13  0  5)
       #( 5  7  9  9  9 14  5  0))
```

```
#(#( 0  3  3  8 13  5  9 13 14)
  #( 3  0  0  6 11  7  7 11 12)
  #( 3  0  0  6 11  7  7 11 12)
  #( 8  6  6  0  5  8 10 13 14)
  #(13 11 11  5  0 12 14 15 17)
  #( 5  7  7  8 12  0 14 18 19)
  #( 9  7  7 10 14 14  0  4  5)
  #(13 11 11 13 15 18  4  0  2)
  #(14 12 12 14 17 19  5  2  0))

#(#( 0  1  5  5  7  8 12 17 15 19)
  #( 1  0  5  5  7  9 13 18 16 20)
  #( 5  5  0 10 12 11 16 22 18 20)
  #( 5  5 10  0  2  9 12 17 16 23)
  #( 7  7 12  2  0 10 11 16 16 23)
  #( 8  9 11  9 10  0  5 10  7 13)
  #(12 13 16 12 11  5  0  5  5 14)
  #(17 18 22 17 16 10  5  0  7 16)
  #(15 16 18 16 16  7  5  7  0  9)
  #(19 20 20 23 23 13 14 16  9  0))

#(#( 0  4  5  6  9 13 13 16 20  7  7)
  #( 4  0  7  9 13 18 17 20 24 10 10)
  #( 5  7  0  2  9 15 15 19 20 10 11)
  #( 6  9  2  0  9 15 15 19 19 11 12)
  #( 9 13  9  9  0  6  6 10 11  7  9)
  #(13 18 15 15  6  0  2  4  7  8 10)
  #(13 17 15 15  6  2  0  4  9  7  8)
  #(16 20 19 19 10  4  4  0  9 10 11)
  #(20 24 20 19 11  7  9  9  0 16 17)
  #( 7 10 10 11  7  8  7 10 16  0  2)
  #( 7 10 11 12  9 10  8 11 17  2  0))

#(#( 0 14 16 19 14 15 18 20 20 22 18 21)
  #(14  0  3 11 10 16 20 22 21 22 17  8)
  #(16  3  0  8  9 15 19 21 19 20 15  9)
  #(19 11  8  0  5 11 13 15 13 13  8 17)
  #(14 10  9  5  0  6 10 12 10 11  6 18)
  #(15 16 15 11  6  0  3  5  5  7  4 24)
  #(18 20 19 13 10  3  0  2  2  4  5 28)
  #(20 22 21 15 12  5  2  0  2  3  6 30)
  #(20 21 19 13 10  5  2  2  0  2  4 29)
  #(22 22 20 13 11  7  4  3  2  0  5 29)
  #(18 17 15  8  6  4  5  6  4  5  0 24)
  #(21  8  9 17 18 24 28 30 29 29 24  0))))
```

```
;;;
;;; Intercity distance matrices for ordered cities
;;;

(define opt-ordered-ic-d-mats
  '#(#()

     #(#(0))

     #(#(0 1)
       #(1 0))

     #(#(0 3 5)
       #(3 0 4)
       #(5 4 0))

     #(#(0 2 4 5)
       #(2 0 2 3)
       #(4 2 0 2)
       #(5 3 2 0))

     #(#(0 3 6 7 7)
       #(3 0 3 5 5)
       #(6 3 0 2 3)
       #(7 5 2 0 2)
       #(7 5 3 2 0))

     #(#( 0  5  7  8 11 16)
       #( 5  0  4  6 10 13)
       #( 7  4  0  2  6 10)
       #( 8  6  2  0  4  8)
       #(11 10  6  4  0  7)
       #(16 13 10  8  7  0))

     #(#( 0  2  5  3  5  2 10)
       #( 2  0  2  4  7  5 10)
       #( 5  2  0  6 10  7 11)
       #( 3  4  6  0  6  5 13)
       #( 5  7 10  6  0  3 11)
       #( 2  5  7  5  3  0  9)
       #(10 10 11 13 11  9  0))

     #(#( 0  4  3  5  8  7  8 10)
       #( 4  0  1  9 13  9 11 11)
       #( 3  1  0  7 11  8 10 11)
       #( 5  9  7  0  5  9  9 14)
       #( 8 13 11  5  0  8  8 13)
       #( 7  9  8  9  8  0  2  5)
       #( 8 11 10  9  8  2  0  5)
       #(10 11 11 14 13  5  5  0))
```

```
#(#( 0  5 13  8  3  3  9 13 14)
  #( 5  0 12  8  7  7 14 18 19)
  #(13 12  0  5 11 11 14 15 17)
  #( 8  8  5  0  6  6 10 13 14)
  #( 3  7 11  6  0  0  7 11 12)
  #( 3  7 11  6  0  0  7 11 12)
  #( 9 14 14 10  7  7  0  4  5)
  #(13 18 15 13 11 11  4  0  2)
  #(14 19 17 14 12 12  5  2  0))

#(#( 0  5  1  5  7  8 12 17 15 19)
  #( 5  0  5 10 12 11 16 22 18 20)
  #( 1  5  0  5  7  9 13 18 16 20)
  #( 5 10  5  0  2  9 12 17 16 23)
  #( 7 12  7  2  0 10 11 16 16 23)
  #( 8 11  9  9 10  0  5 10  7 13)
  #(12 16 13 12 11  5  0  5  5 14)
  #(17 22 18 17 16 10  5  0  7 16)
  #(15 18 16 16 16  7  5  7  0  9)
  #(19 20 20 23 23 13 14 16  9  0))

#(#( 0  4  5  6  9  7  7 13 13 16 20)
  #( 4  0  7  9 13 10 10 17 18 20 24)
  #( 5  7  0  2  9 10 11 15 15 19 20)
  #( 6  9  2  0  9 11 12 15 15 19 19)
  #( 9 13  9  9  0  7  9  6  6 10 11)
  #( 7 10 10 11  7  0  2  7  8 10 16)
  #( 7 10 11 12  9  2  0  8 10 11 17)
  #(13 17 15 15  6  7  8  0  2  4  9)
  #(13 18 15 15  6  8 10  2  0  4  7)
  #(16 20 19 19 10 10 11  4  4  0  9)
  #(20 24 20 19 11 16 17  9  7  9  0))

#(#( 0 15 18 20 20 22 18 14 19 16 14 21)
  #(15  0  3  5  5  7  4  6 11 15 16 24)
  #(18  3  0  2  2  4  5 10 13 19 20 28)
  #(20  5  2  0  2  3  6 12 15 21 22 30)
  #(20  5  2  2  0  2  4 10 13 19 21 29)
  #(22  7  4  3  2  0  5 11 13 20 22 29)
  #(18  4  5  6  4  5  0  6  8 15 17 24)
  #(14  6 10 12 10 11  6  0  5  9 10 18)
  #(19 11 13 15 13 13  8  5  0  8 11 17)
  #(16 15 19 21 19 20 15  9  8  0  3  9)
  #(14 16 20 22 21 22 17 10 11  3  0  8)
  #(21 24 28 30 29 29 24 18 17  9  8  0))))
```

B.4 UNBAL

B.4.1 Dynamic

```
(herald generate-n-tasks)


(lset *tasks-remaining* 0)


(define (main n t)
  (set *task-cycles* t)
  (spawn-and-wait (lambda () (pause t)) n))


(define (spawn-and-wait thunk n)
  (set *tasks-remaining* n)
  (let* ((done (make-placeholder))
         (thunk-1 (lambda ()
                    (thunk)
                    (when (fx<= (modify *tasks-remaining*
                                        (lambda (x) (fx- x 1)))
                                0)
                          (determine done '#t)))))
    (iterate loop ((tasks '()) (i n))
      (if (fx> i 0)
          (loop (cons (make-dummy-task thunk-1) tasks) (fx- i 1))
          (sched-tasks (link-tasks tasks))))
    (touch done)))

(define (make-dummy-task thunk)
  (let ((new-task (make-task))
        (old-task (get-my-task)))
    (when old-task
      (set (task-level new-task) (fx+ (task-level old-task) 1)))
    (set (task-created-on new-task) *my-pid*)
    (set (task-closure new-task)
         (new-task-wrapper new-task thunk '()))
    (stats-creating-task)
    (task-message "Creating  " new-task)
    new-task))

(define (link-tasks tasks)
  (if (null? tasks)
      '()
      (let ((first (car tasks)))
        (iterate loop ((current (car tasks)) (rest (cdr tasks)))
          (cond ((null? rest) first)
                (else
                 (set (task-next current) (car rest))
                 (loop (car rest) (cdr rest))))))))
```

B.4.2 Static

```
(herald generate-n-stat)


(lset *tasks-remaining* 0)

(define (main n t)
  (set *task-cycles* t)
  (spawn-and-wait (lambda () (pause t)) n))


(define (spawn-and-wait thunk n)
  (set *tasks-remaining* n)
  (let* ((done (make-placeholder))
         (n-local (fx/ (fx+ n (fx- *N-Processors* 1)) *N-Processors*))
         (thunk-1 (lambda ()
                    (thunk)
                    (when (fx<= (modify *tasks-remaining*
                                        (lambda (x) (fx- x 1)))
                                0)
                      (determine done '#t)))))
    (do-in-parallel (r c)
      (iterate loop ((tasks '()) (i n-local))
        (if (fx> i 0)
            (loop (cons (make-dummy-task thunk-1) tasks) (fx- i 1))
            (sched-tasks (link-tasks tasks)))))
    (touch done)))

(define (make-dummy-task thunk)
  (let ((new-task (make-task))
        (old-task (get-my-task)))
    (when old-task
      (set (task-level new-task) (fx+ (task-level old-task) 1)))
    (set (task-created-on new-task) *my-pid*)
    (set (task-closure new-task)
         (new-task-wrapper new-task thunk '()))
    (stats-creating-task)
    (task-message "Creating  " new-task)
    new-task))

(define (link-tasks tasks)
  (if (null? tasks)
      '()
      (let ((first (car tasks)))
        (iterate loop ((current (car tasks)) (rest (cdr tasks)))
          (cond ((null? rest) first)
                (else
                 (set (task-next current) (car rest))
                 (loop (car rest) (cdr rest)))))))))
```

B.5 MATMUL

The various versions of MATMUL were actually written as eight nearly identical programs. In this section, we present the code that is common to those programs first. We then separately list the code that contains differences.

B.5.1 Common Code

```
;;;
;;; timing parameters
;;;

(define-constant *loop-cycles*     7)
(define-constant *mul-cycles*     40)
(define-constant *add-cycles*      1)
(define-constant *matref-cycles*   7)
(define-constant *matset-cycles*   8)
(define-constant *lmatref-cycles* 10)
(define-constant *lmatset-cycles* 20)


(define-local-syntax (blocking-forpar header . body)
  (destructure (((name start end) header)
                (loop (generate-symbol 'loop))
                (upper (generate-symbol 'upper))
                (mid (generate-symbol 'mid))
                (pl (generate-symbol 'placeholder)))
    `(iterate ,loop ((,name ,start) (,upper ,end))
       (cond ((fx> ,upper (fx+ ,name 1))
              (let* ((,mid (fx+ ,name (fx-ashr (fx- ,upper ,name) 1)))
                     (,pl (future (,loop ,name ,mid))))
                (,loop ,mid ,upper)
                (touch ,pl)))
             (else ,@body)))))
```

```
;;;
;;; the following code is snarfed from dmatrix.t
;;;

(define-integrable (%my-ceiling x y)
  (fx/ (fx+ x (fx- y 1)) y))

;;; index calculation: more efficient to use FP.
(define (index->block+offset i blocksize)
  ;; returns (fx/ i blocksize), (fx-rem i blocksize) in 40 cycles.
  (let ((fl-i (fixnum->flonum i))
        (fl-blocksize (fixnum->flonum blocksize)))
    (let* ((quotient (flonum->fixnum (fl/ fl-i fl-blocksize)))
           (remainder (fx- i (flonum->fixnum
                               (fl* (fixnum->flonum quotient)
                                    fl-blocksize)))))
      (return quotient remainder))))


;;;
;;; %dmatrix data structure
;;;

(define-structure %dmatrix
  top-matrix
  submat-w
  submat-h)

(define (%make-dmatrix height width make-mat-fn val)
  (let* ((radix *Procs-Per-Dim*)
         (top-matrix (make-matrix radix radix))
         (submat-h (%my-ceiling height radix))
         (submat-w (%my-ceiling width radix)))
    (do-in-parallel (row col)
      (set (MATREF top-matrix row col)
           (make-mat-fn submat-h submat-w val)))
    (let ((dm (make-%dmatrix)))
      (set (%dmatrix-top-matrix dm) (CREATE-DIR-ENTRY top-matrix))
      (set (%dmatrix-submat-h dm) (CREATE-DIR-ENTRY submat-h))
      (set (%dmatrix-submat-w dm) (CREATE-DIR-ENTRY submat-w))
      dm)))
```

```
;;;
;;; MATRIX
;;;

(define (MAKE-MATRIX height width . val)
  (let ((matrix (make-vector height))
        (initval (if val (car val) 0)))
    (do ((row 0 (fx+ row 1)))
        ((fx= row height))
      (let ((vec (make-vector width)))
        (dotimes (i width)
          (set (vref vec i) (CREATE-DIR-ENTRY initval)))
        (set (vref matrix row) (CREATE-DIR-ENTRY vec))))
    matrix))

(define-constant MATREF
  (object (lambda (matrix row col)
            (DIR-READ (vref (DIR-READ (vref matrix row)) col)))
    ((setter self)
     (lambda (matrix row col value)
       (DIR-WRITE (vref (DIR-READ (vref matrix row)) col) value)))))

(define-integrable (MATRIX-HEIGHT m)
  (vector-length m))

(define-integrable (MATRIX-WIDTH m)
  (vector-length (DIR-READ (vref m 0))))
```

```
;;;
;;; LMATRIX (matrix of lstructs)
;;;

(define (MAKE-LMATRIX height width . val)
  (let ((matrix (make-vector height))
        (initval (if val (car val) 0)))
    (do ((row 0 (fx+ row 1)))
        ((fx= row height))
      (let ((vec (make-vector width)))
        (dotimes (i width)
          (let ((pl (make-placeholder)))
            (set (placeholder-determined? pl) '#t)
            (set (placeholder-value pl) initval)
            (set (vref vec i) (CREATE-DIR-ENTRY pl))))
        (set (vref matrix row) (CREATE-DIR-ENTRY vec))))
    matrix))

(define-constant LMATREF
  (object (lambda (matrix row col)
            (let* ((dir (vref (DIR-READ (vref matrix row)) col))
                   (lcell (DIR-READ dir))
                   (val (*lref lcell)))
              (DIR-WRITE dir lcell)
              val))
    ((setter self)
     (lambda (matrix row col value)
            (let* ((dir (vref (DIR-READ (vref matrix row)) col))
                   (lcell (DIR-READ dir)))
              (*l-set lcell value)
              (DIR-WRITE dir lcell)
              value)))))

(define-integrable (LMATRIX-HEIGHT m)
  (vector-length m))

(define-integrable (LMATRIX-WIDTH m)
  (vector-length (DIR-READ (vref m 0))))
```

```
;;;
;;; Distributed matrix
;;;

(define (MAKE-DMATRIX height width . val)
  (%make-dmatrix height width MAKE-MATRIX (if val (car val) 0)))

(define-constant DMATREF
  (object (lambda (dmat row col)
            (let ((sub-h (DIR-READ (%dmatrix-submat-h dmat)))
                  (sub-w (DIR-READ (%dmatrix-submat-w dmat)))
                  (top-matrix (DIR-READ (%dmatrix-top-matrix dmat))))
              (receive (vblock voffset)
                       (index->block+offset row sub-h)
                (receive (hblock hoffset)
                         (index->block+offset col sub-w)
                  (MATREF (MATREF top-matrix vblock hblock)
                          voffset hoffset)))))
    ((setter self)
     (lambda (dmat row col value)
       (let ((sub-h (DIR-READ (%dmatrix-submat-h dmat)))
             (sub-w (DIR-READ (%dmatrix-submat-w dmat)))
             (top-matrix (DIR-READ (%dmatrix-top-matrix dmat))))
         (receive (vblock voffset)
                  (index->block+offset row sub-h)
           (receive (hblock hoffset)
                    (index->block+offset col sub-w)
             (set (MATREF (MATREF top-matrix vblock hblock)
                          voffset hoffset)
                  value))))))))

(define (DMATRIX-SUBMATRIX dm row col)
  (MATREF (DIR-READ (%dmatrix-top-matrix dm)) row col))
```

```
;;;
;;; Distributed l-matrix
;;;

(define (MAKE-DLMATRIX height width . val)
  (%make-dmatrix height width MAKE-LMATRIX (if val (car val) 0)))

(define-constant DLMATREF
  (object (lambda (dmat row col)
            (let ((sub-h (DIR-READ (%dmatrix-submat-h dmat)))
                  (sub-w (DIR-READ (%dmatrix-submat-w dmat)))
                  (top-matrix (DIR-READ (%dmatrix-top-matrix dmat))))
              (receive (vblock voffset)
                       (index->block+offset row sub-h)
                (receive (hblock hoffset)
                         (index->block+offset col sub-w)
                  (LMATREF (MATREF top-matrix vblock hblock)
                           voffset hoffset)))))
    ((setter self)
     (lambda (dmat row col value)
       (let ((sub-h (DIR-READ (%dmatrix-submat-h dmat)))
             (sub-w (DIR-READ (%dmatrix-submat-w dmat)))
             (top-matrix (DIR-READ (%dmatrix-top-matrix dmat))))
         (receive (vblock voffset) (index->block+offset row sub-h)
           (receive (hblock hoffset) (index->block+offset col sub-w)
             (set (LMATREF (MATREF top-matrix vblock hblock)
                           voffset hoffset)
                  value)))))))))

(define (DLMATRIX-SUBMATRIX dm row col)
  (MATREF (DIR-READ (%dmatrix-top-matrix dm)) row col))
```

B.5.2 Cached Versions

These versions of MATMUL simulate coherent full-mapped directories. The code for simulation of cache operations that are kept coherent using those directories is given here.

```
;;;
;;; Coherence protocol constants
;;;

(define-constant *rreq-msg-size* 8)
(define-constant *rresp-msg-size* 24)
(define-constant *wreq-msg-size* 8)
(define-constant *wresp-msg-size* 24)
(define-constant *invr-msg-size* 8)
(define-constant *invr-ack-msg-size* 8)
(define-constant *invw-msg-size* 8)
(define-constant *update-msg-size* 24)

(define-constant *process-rreq-cycles* 4)
(define-constant *process-rresp-cycles* 4)
(define-constant *process-wreq-cycles* 4)
(define-constant *process-wresp-cycles* 4)
(define-constant *process-invr-cycles* 4)
(define-constant *process-invr-ack-cycles* 4)
(define-constant *process-invw-cycles* 4)
(define-constant *process-update-cycles* 4)


;;;
;;; DIR-ENTRY abstraction
;;;

(define-structure DIR-ENTRY
  HOME-PID
  DIRECTORY ;; write: [fixnum] pid
            ;; read:  [list] (len <pid> <pid> ...)
            ;;        [vector] #(<0 has permission>
            ;;                   <1 has permission> ...)
  VALUE
  (((print self port)
    (format port "#{DIR-ENTRY (~s) <~s:~s> ~s}"
            (object-hash self)
            (DIR-ENTRY-HOME-PID self)
            (DIR-ENTRY-DIRECTORY self)
            (DIR-ENTRY-VALUE self)))))
```

```
(define-constant *max-directory-list-length* 16)

(define (CREATE-DIR-ENTRY val)
  (let ((de (make-dir-entry)))
    (set (dir-entry-home-pid de) *my-pid*) ; home node is here
    (set (dir-entry-directory de) *my-pid*) ; I get write permission
    (set (dir-entry-value de) val)
    de))

(define-integrable (DIR-READ x)
  (get-read-permission x)
  (DIR-ENTRY-VALUE x))

(define-integrable (DIR-WRITE x val)
  (get-write-permission x)
  (set (DIR-ENTRY-VALUE x) val))


(define-integrable (pid->dir-index pid)
  (fx-ashr pid 4))

(define-integrable (pid->dir-bit pid)
  (fx-ashl 1 (fx-and pid #xf)))


(define-integrable (I-have-write-permission x)
  (has-write-permission *my-pid* x))

(define-integrable (has-write-permission pid x)
  (let ((dir (dir-entry-directory x)))
    (and (fixnum? dir) (fx= dir pid))))

(define-integrable (add-write-permission pid x)
  (set (dir-entry-directory x) pid))
```

```
(define-integrable (I-have-read-permission x)
  (let ((dir (dir-entry-directory x)))
    (or (and (list? dir) (has-read-permission-list *my-pid* x))
        (and (vector? dir) (has-read-permission-vector *my-pid* x))
        (I-have-write-permission x))))

(define-integrable (has-read-permission-list pid x)
  (%has-read-permission-list pid (dir-entry-directory x)))

(define-integrable (%has-read-permission-list pid l)
  (memq? pid (cdr l)))

(define-integrable (has-read-permission-vector pid x)
  (%has-read-permission-vector pid (dir-entry-directory x)))

(define-integrable (%has-read-permission-vector pid vec)
  (fxn= (fx-and (vref vec (pid->dir-index pid))
                (pid->dir-bit pid))
        0))

(define-integrable (add-read-permission-list pid x)
  (let ((l (dir-entry-directory x)))
    (set (cdr l) (cons pid (cdr l)))
    (set (car l) (fx+ (car l) 1))))

(define-integrable (add-read-permission-vector pid x)
  (let ((vec (dir-entry-directory x))
        (index (pid->dir-index pid)))
    (set (vref vec index)
         (fx-ior (vref vec index) (pid->dir-bit pid)))))
```

```
(define-integrable (get-read-permission x)
  (cond ((I-have-read-permission x))
        ((fixnum? (dir-entry-directory x)) ; someone else has write
         (get-read-permission-from-write-state x)) ;     permission
        ((vector? (dir-entry-directory x))
         (get-read-permission-from-read-state-vector x))
        (else
         (get-read-permission-from-read-state-list x))))

(define (get-read-permission-from-write-state x)
  (let ((fpid (dir-entry-directory x)))
    (set (dir-entry-directory x) (cons 1 (cons *my-pid* '())))
    (pause-read-from-write-time x fpid)))

(define (get-read-permission-from-read-state-vector x)
  (add-read-permission-vector *my-pid* x)
  (pause-read-from-read-time x))

(define (get-read-permission-from-read-state-list x)
  (cond ((fx>= (car (dir-entry-directory x))
               *max-directory-list-length*)
         (read-directory-list->vector x)
         (get-read-permission-from-read-state-vector x))
        (else
         (add-read-permission-list *my-pid* x)
         (pause-read-from-read-time x))))

(define (read-directory-list->vector x)
  (let ((l (dir-entry-directory x))
        (v (make-vector (fx-ashr *n-processors* 4))))
    (set (dir-entry-directory x) v)
    (dolist (pid (cdr l))
      (add-read-permission-vector pid x))))
```

```
(define (pause-read-from-read-time x)
  (let ((hpid (dir-entry-home-pid x)))
    (pause (+ (transit-time *rreq-msg-size* *my-pid* hpid)
              *process-rreq-cycles*
              (transit-time *rresp-msg-size* hpid *my-pid*)
              *process-rresp-cycles*))))

(define (pause-read-from-write-time x fpid)
  (let ((hpid (dir-entry-home-pid x)))
    (pause (+ (transit-time *rreq-msg-size* *my-pid* hpid)
              *process-rreq-cycles*
              (transit-time *invw-msg-size* hpid fpid)
              *process-invw-cycles*
              (transit-time *update-msg-size* fpid hpid)
              *process-update-cycles*
              (transit-time *rresp-msg-size* hpid *my-pid*)
              *process-rresp-cycles*))))
```

```
(define-integrable (get-write-permission x)
  (cond ((I-have-write-permission x))
        ((fixnum? (dir-entry-directory x)) ; someone else has write
         (get-write-permission-from-write-state x)) ;    permission
        ((vector? (dir-entry-directory x))
         (get-write-permission-from-read-state-vector x))
        (else
         (get-write-permission-from-read-state-list x))))

(define (get-write-permission-from-write-state x)
  (let ((fpid (dir-entry-directory x)))
    (add-write-permission *my-pid* x)
    (pause-write-from-write-time x fpid)))

(define (get-write-permission-from-read-state-vector x)
  (let ((dir (dir-entry-directory x)))
    (add-write-permission *my-pid* x)
    (pause-write-from-read-time-vector x dir)))

(define (get-write-permission-from-read-state-list x)
  (let ((dir (dir-entry-directory x)))
    (add-write-permission *my-pid* x)
    (pause-write-from-read-time-list x dir)))
```

```
(define (pause-write-from-write-time x fpid)
  (let ((hpid (dir-entry-home-pid x)))
    (pause (+ (transit-time *wreq-msg-size* *my-pid* hpid)
              *process-wreq-cycles*
              (transit-time *invw-msg-size* hpid fpid)
              *process-invw-cycles*
              (transit-time *update-msg-size* fpid hpid)
              *process-update-cycles*
              (transit-time *wresp-msg-size* hpid *my-pid*)
              *process-wresp-cycles*))))

(define-integrable (pause-write-from-read-time-vector x vec)
  (pause-write-from-read-time x vec %has-read-permission-vector))

(define-integrable (pause-write-from-read-time-list x l)
  (pause-write-from-read-time x l %has-read-permission-list))

(define (pause-write-from-read-time x dir permission-fn)
  (let ((hpid (dir-entry-home-pid x)))
    (pause (+ (transit-time *wreq-msg-size* *my-pid* hpid)
              *process-wreq-cycles*
              (transit-time *wresp-msg-size* hpid *my-pid*)
              *process-wresp-cycles*))
    (iterate loop ((pid 0) (n 0) (max-dist 0) (max-pid hpid))
      (cond ((fx>= pid *N-Processors*)
             (pause (+ (transit-time *invr-msg-size* hpid max-pid)
                       *process-invr-cycles*
                       (transit-time *invr-ack-msg-size* max-pid hpid)
                       *process-invr-ack-cycles*
                       (* (fx- n 1) *invr-msg-size*))))
            ((permission-fn pid dir)
             (let ((dist (node-distance hpid pid)))
               (if (fx<= dist max-dist)
                   (loop (fx+ pid 1) (fx+ n 1) max-dist max-pid)
                   (loop (fx+ pid 1) (fx+ n 1) dist pid))))
            (else
             (loop (fx+ pid 1) n max-dist max-pid))))))
```

Coarse-Grained, Dynamic, With Caches

```
(define (main i j k)
  (let ((m1 (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 m3)
    m3))

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dim*))
    (blocking-forpar (i 0 n)
      (blocking-forpar (j 0 n)
        (matmul-row-col i j m1 m2 m3 n)))))

(define (matmul-row-col i j m1 m2 m3 n)
  (for (k 0 n)
    (matmul-blocks (DMATRIX-SUBMATRIX m1 i k)
                   (DMATRIX-SUBMATRIX m2 k j)
                   (DMATRIX-SUBMATRIX m3 i j))))

(define (matmul-blocks x y z)
  (let ((x-height (MATRIX-HEIGHT x))
        (x-width (MATRIX-WIDTH x))
        (y-width (MATRIX-WIDTH y)))
    (dotimes (i x-height)
      (pause *loop-cycles*)
      (dotimes (j y-width)
        (pause *loop-cycles*)
        (set (MATREF z i j)
             (fx+ (MATREF z i j)
                  (acc (k 0 x-width)
                       (pause (+ *mul-cycles* *add-cycles*
                                 *loop-cycles*))
                       (fx* (MATREF x i k) (MATREF y k j)))))
        (pause (+ *matref-cycles* *add-cycles* *matset-cycles*))))))
```

Coarse-Grained, Static, With Caches

```
(define (main i j k)
  (let ((m1 (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 m3)
    m3))

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dim*))
    (do-in-parallel (i j)
      (for (k 0 n)
        (matmul-blocks (DMATRIX-SUBMATRIX m1 i k)
                       (DMATRIX-SUBMATRIX m2 k j)
                       (DMATRIX-SUBMATRIX m3 i j))))))

(define (matmul-blocks x y z)
  (let ((x-height (MATRIX-HEIGHT x))
        (x-width (MATRIX-WIDTH x))
        (y-width (MATRIX-WIDTH y)))
    (dotimes (i x-height)
      (pause *loop-cycles*)
      (dotimes (j y-width)
        (pause *loop-cycles*)
        (set (MATREF z i j)
             (fx+ (MATREF z i j)
                  (acc (k 0 x-width)
                       (pause (+ *mul-cycles* *add-cycles*
                                 *loop-cycles*))
                       (fx* (MATREF x i k) (MATREF y k j)))))
        (pause (+ *matref-cycles* *add-cycles* *matset-cycles*))))))
```

Fine-Grained, Dynamic, With Caches

```
(define (main i j k)
  (let ((m1 (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DLMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 m3)
    m3))

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dim*))
    (blocking-forpar (i 0 n)
      (blocking-forpar (j 0 n)
        (blocking-forpar (k 0 n)
          (matmul-blocks (DMATRIX-SUBMATRIX m1 i k)
                         (DMATRIX-SUBMATRIX m2 k j)
                         (DLMATRIX-SUBMATRIX m3 i j)))))))

(define (matmul-blocks x y z)
  (let ((x-height (MATRIX-HEIGHT x))
        (x-width (MATRIX-WIDTH x))
        (y-width (MATRIX-WIDTH y)))
    (dotimes (i x-height)
      (pause *loop-cycles*)
      (dotimes (j y-width)
        (pause *loop-cycles*)
        (set (LMATREF z i j)
             (fx+ (LMATREF z i j)
                  (acc (k 0 x-width)
                       (pause (+ *mul-cycles* *add-cycles*
                                 *loop-cycles*))
                       (fx* (MATREF x i k) (MATREF y k j)))))
        (pause (+ *lmatref-cycles* *add-cycles* *lmatset-cycles*))))))
```

Fine-Grained, Static, With Caches

```
(define (main i j k)
  (let ((m1 (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DLMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 m3)
    m3))

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dim*))
    (do-in-parallel (i j)
      (blocking-forpar (k 0 n)
        (matmul-blocks (DMATRIX-SUBMATRIX m1 i k)
                       (DMATRIX-SUBMATRIX m2 k j)
                       (DLMATRIX-SUBMATRIX m3 i j))))))

(define (matmul-blocks x y z)
  (let ((x-height (MATRIX-HEIGHT x))
        (x-width (MATRIX-WIDTH x))
        (y-width (MATRIX-WIDTH y)))
    (dotimes (i x-height)
      (pause *loop-cycles*)
      (dotimes (j y-width)
        (pause *loop-cycles*)
        (set (LMATREF z i j)
             (fx+ (LMATREF z i j)
                  (acc (k 0 x-width)
                       (pause (+ *mul-cycles* *add-cycles*
                                 *loop-cycles*))
                       (fx* (MATREF x i k) (MATREF y k j)))))
        (pause (+ *lmatref-cycles* *add-cycles* *lmatset-cycles*))))))
```

B.5.3 Uncached Versions

These versions of MATMUL do not simulate caches for global memory. The caching code given above is replaced by the following macros.

```
;;;
;;; No Coherent Caches -- reads and writes of shared structures always
;;;  generate reads and writes over the network.
;;;

(define-local-syntax (DIR-READ x)
  (let ((data (generate-symbol 'data))
        (pid  (generate-symbol 'pid))
        (rval (generate-symbol 'rval)))
    `(let* ((,data ,x)
            (,pid (DIR-ENTRY-HOME-PID ,data))
            (,rval (cond ((fx= ,pid *my-pid*)
                          (pause *process-rreq-cycles*)
                          (DIR-ENTRY-VALUE ,data))
                         (else
                          (remote-access
                           ,pid
                           *rreq-msg-size*
                           *process-rreq-cycles*
                           *rresp-msg-size*
                           (DIR-ENTRY-VALUE ,data))))))
       (pause *process-rresp-cycles*)
       ,rval)))

(define-local-syntax (DIR-WRITE x val)
  (let ((data (generate-symbol 'data))
        (pid  (generate-symbol 'pid))
        (wval (generate-symbol 'wval)))
    `(let* ((,data ,x)
            (,wval (cond ((fx= ,pid *my-pid*)
                          (pause *process-wreq-cycles*)
                          (set (DIR-ENTRY-VALUE ,data) ,val))
                         (else
                          (remote-access
                           ,pid
                           *wreq-msg-size*
                           *process-wreq-cycles*
                           *wresp-msg-size*
                           (set (DIR-ENTRY-VALUE ,data) ,val))))))
       (pause *process-wresp-cycles*)
       ,wval)))
```

Coarse-Grained, Dynamic, No Caches

```
(define (main i j k)
  (let ((m1 (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 m3)
    m3))

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dim*))
    (blocking-forpar (i 0 n)
      (blocking-forpar (j 0 n)
        (matmul-row-col i j m1 m2 m3 n)))))

(define (matmul-row-col i j m1 m2 m3 n)
  (for (k 0 n)
    (matmul-blocks (DMATRIX-SUBMATRIX m1 i k)
                   (DMATRIX-SUBMATRIX m2 k j)
                   (DMATRIX-SUBMATRIX m3 i j))))

(define (matmul-blocks x y z)
  (let ((x-height (MATRIX-HEIGHT x))
        (x-width (MATRIX-WIDTH x))
        (y-width (MATRIX-WIDTH y)))
    (dotimes (i x-height)
      (pause *loop-cycles*)
      (dotimes (j y-width)
        (pause *loop-cycles*)
        (set (MATREF z i j)
             (fx+ (MATREF z i j)
                  (acc (k 0 x-width)
                       (pause (+ *mul-cycles* *add-cycles*
                                 *loop-cycles*))
                       (fx* (MATREF x i k) (MATREF y k j)))))
        (pause (+ *matref-cycles* *add-cycles* *matset-cycles*))))))
```

Coarse-Grained, Static, No Caches

```
(define (main i j k)
  (let ((m1 (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 m3)
    m3))

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dim*))
    (do-in-parallel (i j)
      (for (k 0 n)
        (matmul-blocks (DMATRIX-SUBMATRIX m1 i k)
                       (DMATRIX-SUBMATRIX m2 k j)
                       (DMATRIX-SUBMATRIX m3 i j))))))

(define (matmul-blocks x y z)
  (let ((x-height (MATRIX-HEIGHT x))
        (x-width (MATRIX-WIDTH x))
        (y-width (MATRIX-WIDTH y)))
    (dotimes (i x-height)
      (pause *loop-cycles*)
      (dotimes (j y-width)
        (pause *loop-cycles*)
        (set (MATREF z i j)
             (fx+ (MATREF z i j)
                  (acc (k 0 x-width)
                       (pause (+ *mul-cycles* *add-cycles*
                                 *loop-cycles*))
                       (fx* (MATREF x i k) (MATREF y k j)))))
        (pause (+ *matref-cycles* *add-cycles* *matset-cycles*))))))
```

Fine-Grained, Dynamic, No Caches

```
(define (main i j k)
  (let ((m1 (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DLMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 m3)
    m3))

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dim*))
    (blocking-forpar (i 0 n)
      (blocking-forpar (j 0 n)
        (blocking-forpar (k 0 n)
          (matmul-blocks (DMATRIX-SUBMATRIX m1 i k)
                         (DMATRIX-SUBMATRIX m2 k j)
                         (DLMATRIX-SUBMATRIX m3 i j)))))))

(define (matmul-blocks x y z)
  (let ((x-height (MATRIX-HEIGHT x))
        (x-width (MATRIX-WIDTH x))
        (y-width (MATRIX-WIDTH y)))
    (dotimes (i x-height)
      (pause *loop-cycles*)
      (dotimes (j y-width)
        (pause *loop-cycles*)
        (set (LMATREF z i j)
             (fx+ (LMATREF z i j)
                  (acc (k 0 x-width)
                       (pause (+ *mul-cycles* *add-cycles*
                                 *loop-cycles*))
                       (fx* (MATREF x i k) (MATREF y k j)))))
        (pause (+ *lmatref-cycles* *add-cycles* *lmatset-cycles*))))))
```

Fine-Grained, Static, No Caches

```
(define (main i j k)
  (let ((m1 (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DLMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 m3)
    m3))

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dim*))
    (do-in-parallel (i j)
      (blocking-forpar (k 0 n)
        (matmul-blocks (DMATRIX-SUBMATRIX m1 i k)
                       (DMATRIX-SUBMATRIX m2 k j)
                       (DLMATRIX-SUBMATRIX m3 i j))))))

(define (matmul-blocks x y z)
  (let ((x-height (MATRIX-HEIGHT x))
        (x-width (MATRIX-WIDTH x))
        (y-width (MATRIX-WIDTH y)))
    (dotimes (i x-height)
      (pause *loop-cycles*)
      (dotimes (j y-width)
        (pause *loop-cycles*)
        (set (LMATREF z i j)
             (fx+ (LMATREF z i j)
                  (acc (k 0 x-width)
                       (pause (+ *mul-cycles* *add-cycles*
                                 *loop-cycles*))
                       (fx* (MATREF x i k) (MATREF y k j)))))
        (pause (+ *lmatref-cycles* *add-cycles* *lmatset-cycles*))))))
```

Appendix C

Raw Data

The following tables list the running times for the various applications run under the various thread managers for the machine sizes and network speeds given.

C.1 AQ

C.1.1 $t_n = 1$ Cycle / Flit-Hop

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	435035	112062	34129	15087	14005	14005	14005	14005
P-Ideal	437191	118076	36085	18346	16616	16627	16627	16627
C-Ideal-1	437209	115735	35832	18254	16467	16970	17304	17432
C-Ideal-2	437209	114419	36058	16989	16942	16875	17279	17416
RR-1	437191	117456	40417	19950	17684	18907	18694	– –
RR-2	437191	116304	37367	21135	17932	18534	18539	– –
Diff-1	445119	126182	49405	44363	45505	45523	– –	– –
Diff-2	445119	126875	46455	33036	34017	34019	– –	– –
TTM	437191	119230	39386	21287	18493	18410	18385	20405
XTM	437191	116045	40749	22285	25064	24280	21566	21999
XTM-C	437191	121966	47901	26724	31433	32781	29542	29138

Table C.1: *AQ(0.5) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	2131170	542674	141886	43030	19226	16149	16149	16149
P-Ideal	2141754	564540	150438	45283	22013	19394	19443	19443
C-Ideal-1	2141772	542714	145202	47139	23980	19464	20160	20465
C-Ideal-2	2141772	542502	145551	46704	22398	19618	20188	20801
RR-1	2141754	545377	150423	53676	28722	28662	27506	– –
RR-2	2141754	544633	149842	52986	33086	30014	26958	– –
Diff-1	2180964	592222	174096	89890	100840	100836	– –	– –
Diff-2	2180964	609435	173445	68067	62344	63054	– –	– –
TTM	2141754	551045	156336	53321	28386	24167	25213	25497
XTM	2141754	550285	152832	54432	39297	29941	27228	31589
XTM-C	2147417	560526	158993	60323	54796	49488	72479	56160

Table C.2: *AQ(0.1) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	4148500	1054102	270814	73986	27454	16150	16150	16149
P-Ideal	4180434	1095981	283181	77950	30024	19828	18991	19330
C-Ideal-1	4169126	1050843	274270	78660	30961	19750	20033	20417
C-Ideal-2	4169126	1049507	273153	79023	31423	19869	19759	20818
RR-1	4169108	1055143	279758	89091	40828	35272	33974	– –
RR-2	4169108	1054012	276636	87606	38038	38258	40102	– –
Diff-1	4244440	1147785	310130	119954	125941	122621	– –	– –
Diff-2	4244440	1178724	320553	102799	101305	86826	– –	– –
TTM	4180434	1066320	284062	91730	38763	29169	25684	25957
XTM	4169108	1055502	287066	87052	50737	33773	34173	33661
XTM-C	4169108	1076687	303061	97568	61870	57997	68921	66078

Table C.3: *AQ(0.05) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	20101185	5092266	1292960	333877	92863	31235	20438	20438
P-Ideal	20212387	5298344	1346799	344455	96475	36430	25419	25218
C-Ideal-1	20201079	5059478	1282056	341638	98190	38078	26236	26977
C-Ideal-2	20201079	5063978	1280377	333620	97802	40671	25788	26697
RR-1	20201061	5062577	1287129	356119	116679	81243	81555	– –
RR-2	20201061	5062895	1286878	351224	119718	77522	73782	– –
Diff-1	20562663	5531705	1448165	410432	297889	316372	– –	– –
Diff-2	20562663	5690009	1508596	406821	185665	176581	– –	– –
TTM	20201061	5083434	1323368	365086	111199	55613	36051	37040
XTM	20201061	5080273	1309375	359148	121911	64274	47564	42096
XTM-C	20201061	5155594	1363204	391722	158337	81674	80557	86459

Table C.4: *AQ(0.01) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	42849680	10853983	2753885	702964	184603	56967	25354	23975
P-Ideal	43073918	11294466	2856836	723780	191625	62944	30202	25601
C-Ideal-1	43062610	10776279	2717726	710372	189768	62878	32043	26622
C-Ideal-2	43068273	10781077	2715584	698536	189189	63505	32768	26708
RR-1	43062592	10780173	2720015	726801	223000	125034	117578	– –
RR-2	43062592	10792555	2728316	723552	210191	104606	105919	– –
Diff-1	43832400	11785704	3060293	806422	449619	441495	– –	– –
Diff-2	43832400	12127888	3201159	851806	276455	241594	– –	– –
TTM	43062592	10802275	2763857	746424	214560	84572	51378	41642
XTM	43062592	10809656	2763583	731563	226962	105396	64571	62185
XTM-C	43068255	10974037	2877550	800739	312924	141977	100445	131108

Table C.5: *AQ(0.005) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	212564610	53811875	13609923	3448238	878585	231076	69970	29874
P-Ideal	213632152	55949997	14130644	3554005	900669	239586	74201	34876
C-Ideal-1	213609518	53409304	13379388	3442923	877465	238533	78261	38143
C-Ideal-2	213615181	53436659	13389600	3371806	864727	238743	78432	38528
RR-1	213620826	53416489	13385074	3423732	954951	403871	326324	– –
RR-2	213609500	53438267	13406936	3467763	934271	328244	280841	– –
Diff-1	217412192	58408211	15120932	3888689	1080120	916236	– –	– –
Diff-2	217440993	60112951	15824024	4111792	1094066	640743	– –	– –
TTM	213632152	53479234	13505753	3549928	950098	301909	118771	63726
XTM	213626489	53500740	13502898	3448766	932373	345939	157037	96147
XTM-C	213615163	54176113	14022452	3741867	1121419	426006	202913	158011

Table C.6: *AQ(0.001) – Running Times (cycles).*

C.1.2 Variable t_n

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	20101185	5092493	1293694	333288	93357	31832	20537	20537
P-Ideal	20212387	5299453	1344679	346903	99621	36910	24814	24775
C-Ideal-1	20201097	5059462	1283266	334912	98373	39536	26821	27556
C-Ideal-2	20201097	5064448	1279749	333313	97137	41090	27873	27584
RR-1	20201061	5062318	1301370	383990	153659	99295	97459	– –
RR-2	20201061	5067324	1297694	367722	140735	104137	97089	– –
Diff-1	20562131	5873382	1573857	445112	325325	305601	– –	– –
Diff-2	20562131	6046213	1636698	451022	250021	247527	– –	– –
TTM	20201061	5079447	1324433	362120	120322	53782	39983	41869
XTM	20201061	5079338	1310847	352478	126063	66980	53855	46578
XTM-C	20206724	5157174	1378281	399026	159428	72177	87160	127863

Table C.7: *AQ(0.01) – t_n = 2 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	20101211	5092227	1294561	332725	91614	34278	20743	20743
P-Ideal	20212423	5303939	1345158	349025	98403	38643	25714	25778
C-Ideal-1	20212477	5061063	1285544	336478	99937	40093	28199	29359
C-Ideal-2	20201151	5065698	1283590	333864	98301	41170	27550	30351
RR-1	20201113	5066859	1308516	376101	163530	122435	144460	– –
RR-2	20201113	5065651	1313859	375094	175093	150910	164107	– –
Diff-1	20563652	6565006	1829308	528693	352323	377463	– –	– –
Diff-2	20563652	6732366	1893449	524699	306926	309844	– –	– –
TTM	20201197	5085276	1330267	382041	126112	60398	44935	43657
XTM	20201235	5073495	1309611	355473	120086	70032	47941	46875
XTM-C	20201153	5148725	1370046	396630	146105	80171	84630	93370

Table C.8: *AQ(0.01) – t_n = 4 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	20101263	5089159	1296128	336103	92870	33446	21237	21237
P-Ideal	20206850	5305578	1347103	346523	98524	40262	28304	27134
C-Ideal-1	20201223	5060382	1287232	338458	104206	45293	31178	36547
C-Ideal-2	20201223	5066227	1287395	339847	104544	43762	31813	38917
RR-1	20201820	5089136	1321956	439734	226899	188339	196214	– –
RR-2	20201191	5100399	1333196	383770	213222	198026	199654	– –
Diff-1	20562577	7935005	2334166	673266	440028	468642	– –	– –
Diff-2	20568330	8080183	2388992	693360	478788	442468	– –	– –
TTM	20206986	5093238	1346358	378601	139401	78497	54012	67661
XTM	20201343	5076740	1309959	359356	146317	72398	54588	54363
XTM-C	20201225	5157686	1362555	399678	142857	87429	82477	85041

Table C.9: *AQ(0.01) – t_n = 8 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	20101367	5087882	1289253	330762	93377	35005	21432
P-Ideal	20201349	5313573	1349120	348855	100552	40071	28722
C-Ideal-1	20201385	5061883	1296083	340302	105081	47603	38895
C-Ideal-2	20212711	5073267	1287423	342404	107550	49316	41389
RR-1	20202285	5094770	1330448	480289	272499	276916	254097
RR-2	20201347	5129123	1353045	461425	317963	313609	304211
Diff-1	20562599	10683379	3333567	994517	702582	687026	– –
Diff-2	20574141	10838485	3401989	1036899	685123	657468	– –
TTM	20213124	5110798	1374152	423894	173319	93025	87635
XTM	20224039	5101166	1323069	382214	180691	94893	84169
XTM-C	20201387	5162093	1368166	439721	170311	93938	93656

Table C.10: *AQ(0.01) – t_n = 16 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	20197722	5099148	1292733	335858	95552	36017	26176
P-Ideal	20300389	5279082	1344262	354152	112299	59733	55886
C-Ideal-1	20300533	5098486	1336288	388929	133236	83669	92458
C-Ideal-2	20300533	5112646	1318222	382564	144596	90679	115074
RR-1	20300435	5219740	1599019	808819	633466	634270	615613
RR-2	20300435	5219832	1550463	808250	593332	658476	683631
Diff-1	20804163	27373382	9347160	3095301	2713481	2643436	– –
Diff-2	20804163	27637925	9500319	3103397	2176886	2129328	– –
TTM	20301711	5163685	1457252	464483	273988	207599	192060
XTM	20306500	5209530	1396477	470213	236521	152687	158708
XTM-C	20300837	5194643	1464410	495743	283201	263703	358852

Table C.11: *AQ(0.01) – t_n = 64 Cycles / Flit-Hop – Running Times (cycles).*

C.2 FIB

C.2.1 $t_n = 1$ Cycle / Flit-Hop

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	635387	157638	40751	12845	6451	5413	5412	5412
P-Ideal	652439	181509	50068	16011	9000	7615	7597	7615
C-Ideal-1	652457	168317	47300	17914	10660	9565	10007	10601
C-Ideal-2	652457	168714	47871	18885	10564	9907	9701	10493
RR-1	652439	168910	51950	23679	19527	17932	18455	– –
RR-2	652439	169247	51134	23742	19366	20220	20292	– –
Diff-1	664725	187087	59471	40830	40976	40976	– –	– –
Diff-2	664725	188783	59769	29616	30894	29412	– –	– –
TTM	652439	176628	56311	25182	18826	18883	19984	17928
XTM	652439	174038	54483	23898	20975	18556	20204	17600
XTM-C	652439	184850	67767	33709	33980	30217	53639	57930

Table C.12: *FIB(15) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	7055052	1745573	429066	110201	31113	12109	7712	7353
P-Ideal	7244444	1988825	517631	135793	39242	16469	10926	10432
C-Ideal-1	7244462	1816654	461306	125929	40277	19130	14831	15353
C-Ideal-2	7244462	1821342	464670	126670	40658	18865	15266	15659
RR-1	7244444	1818304	471419	142426	62004	52717	51408	– –
RR-2	7244444	1820234	484434	147437	73138	58001	54714	– –
Diff-1	7373624	1981124	526230	160245	119901	112094	– –	– –
Diff-2	7373624	2044501	552452	164149	94144	92985	– –	– –
TTM	7244444	1837674	494438	146775	62914	40510	29899	29988
XTM	7244444	1845089	487584	155452	76146	44160	30403	34245
XTM-C	7244444	1909502	574192	193382	96755	73132	156931	224980

Table C.13: *FIB(20) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	78250232	19331725	4740547	1179706	300767	80132	24342	11750
P-Ideal	80350904	21979404	5697561	1444026	368877	100119	33108	17178
C-Ideal-1	80350922	20095167	5036588	1282851	337300	96459	37280	23350
C-Ideal-2	80350922	20106973	5045701	1280106	338657	97064	38695	24434
RR-1	80350904	20096730	5052170	1317302	406706	187035	178050	– –
RR-2	80350904	20108332	5108362	1353515	434266	233167	222011	– –
Diff-1	81777562	21829318	5678117	1469149	463284	437829	– –	– –
Diff-2	81777562	22609164	5959726	1562207	448102	343993	– –	– –
TTM	80350904	20141997	5169924	1392524	437969	160430	72452	48338
XTM	80350904	20152893	5158704	1390949	444279	205754	99185	72473
XTM-C	80350904	24481883	5845312	1727970	609347	256368	399127	453817

Table C.14: *FIB(25) – Running Times (cycles).*

C.2.2 Variable t_n

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	7055052	1748341	432185	110991	31265	11900	7559	7546
P-Ideal	7244444	1986597	518016	135509	40521	17006	11760	11202
C-Ideal-1	7244462	1818527	464326	126800	40595	21158	16430	18562
C-Ideal-2	7244462	1820591	466078	127340	42450	21794	16587	17976
RR-1	7244444	1822291	474991	145363	73299	60292	63251	– –
RR-2	7244444	1828478	478125	150398	84345	71391	75580	– –
Diff-1	7373688	2107538	572836	176241	148413	151513	– –	– –
Diff-2	7373688	2167362	592608	171304	116465	103883	– –	– –
TTM	7244444	1841130	491086	153799	67428	42574	37743	39717
XTM	7244444	1867459	494315	151894	60812	44486	34242	30249
XTM-C	7244444	1961428	597400	206157	106275	86053	178461	171457

Table C.15: *FIB(20) – t_n = 2 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	7055078	1759182	432740	110666	31683	11952	8003	7912
P-Ideal	7244480	2010998	523937	139389	41342	17634	12827	12977
C-Ideal-1	7244498	1819160	464584	128870	42968	23543	20391	25006
C-Ideal-2	7244498	1821243	468578	126515	43370	24341	20821	25325
RR-1	7244470	1824799	474802	152429	87893	80588	79072	– –
RR-2	7244470	1837599	480901	161541	119573	110197	112267	– –
Diff-1	7408048	2374267	671046	208648	160240	163791	– –	– –
Diff-2	7408048	2424098	693433	205738	128446	130195	– –	– –
TTM	7244562	1837908	497081	163188	69283	48931	38727	46445
XTM	7244572	1821505	481610	150663	75886	51016	44460	38472
XTM-C	7244490	1943942	558931	197515	123996	118058	194132	170883

Table C.16: *FIB(20) – t_n = 4 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	7055130	1780261	433159	111205	31719	13377	8936	8872
P-Ideal	7244516	2019017	527771	140046	42754	20686	15818	16647
C-Ideal-1	7244534	1820232	465783	133385	47757	26280	26155	33170
C-Ideal-2	7244534	1826416	467037	130588	48097	30072	27037	39954
RR-1	7244522	1827965	490870	179041	108505	113288	108732	– –
RR-2	7244522	1850326	508744	210175	171235	165516	157364	– –
Diff-1	7373992	2862551	858206	281516	213001	208463	– –	– –
Diff-2	7373992	2917805	875363	293488	240687	236879	– –	– –
TTM	7244652	1848180	524020	181886	80810	65572	78751	70533
XTM	7244662	1836490	495480	159742	70338	53176	40310	47491
XTM-C	7244544	1936986	604019	210688	119359	141378	198389	270567

Table C.17: *FIB(20) – t_n = 8 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	7055234	1798918	452382	117620	34243	14341	10598
P-Ideal	7244624	1921304	500173	135957	45422	25199	22033
C-Ideal-1	7244642	1824620	469412	133447	54192	34348	42492
C-Ideal-2	7244642	1836399	475337	138790	56358	41800	56770
RR-1	7244626	1868163	523030	212272	159375	148379	153555
RR-2	7244626	1934930	620762	332189	275455	250806	222271
Diff-1	7373786	3852791	1243938	420987	348716	327626	– –
Diff-2	7373786	3902435	1249959	403378	311929	307490	– –
TTM	7244850	1878020	535304	191020	111846	95326	117295
XTM	7244850	1855064	537298	202074	103859	74319	66744
XTM-C	7244634	1947001	600314	278499	305830	244221	390895

Table C.18: *FIB(20) – t_n = 16 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	7130049	1834009	467888	128719	42126	21906	21585
P-Ideal	7324486	1865705	499085	156675	82806	88391	77426
C-Ideal-1	7316384	1853647	493737	151475	79889	69690	97937
C-Ideal-2	7316384	1866739	502636	160024	82932	76475	112934
RR-1	7324510	1958795	699563	420188	352456	330024	345677
RR-2	7324510	2015863	688651	440963	476833	422665	435985
Diff-1	7425235	9997839	3553327	1417313	1256328	1249922	– –
Diff-2	7425235	10069369	3524931	1321262	1144075	1132898	– –
TTM	7325460	1905376	618049	284587	179006	198796	240065
XTM	7331310	1868209	587810	229439	156843	128660	123495

Table C.19: *FIB(20) – t_n = 64 Cycles / Flit-Hop – Running Times (cycles).*

C.3 TSP

C.3.1 $t_n = 1$ Cycle / Flit-Hop

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	3514490	543917	144389	51611	16562	11679	11643
P-Ideal	3635870	631272	157874	47468	20371	14114	17196
C-Ideal-1	3635888	589285	148815	49261	22223	14651	18501
C-Ideal-2	3635888	598458	147554	47614	20880	14809	18059
RR-1	3635870	579768	152304	56602	33108	28385	30616
RR-2	3635870	590096	154367	55765	31167	29900	32736
Diff-1	3701646	799186	170323	73824	70035	68972	– –
Diff-2	3701646	622469	175249	62027	65226	64838	– –
TTM	3635870	785756	156168	55840	29353	24169	20781
XTM	3635870	599244	158474	58582	39826	34023	30325
XTM-C	3635870	805614	171855	74057	50162	40302	49346

Table C.20: *TSP(8) – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	12188860	2810166	600440	179353	50518	19740	24325
P-Ideal	12613984	3313219	910147	222944	61708	22862	24671
C-Ideal-1	12614002	3257010	867710	218874	60053	24028	25192
C-Ideal-2	12614002	3068803	813595	222760	63100	24771	26176
RR-1	12613984	3254456	877521	237613	89107	64305	61712
RR-2	12613984	3125429	829885	243429	94477	59541	66103
Diff-1	12839426	3633033	1048051	259083	151099	159196	– –
Diff-2	12839426	3189390	966454	265024	133908	135128	– –
TTM	12613984	3651367	862041	226416	80278	40875	31815
XTM	12613984	3012289	845888	242000	82588	60266	39302
XTM-C	12613984	3824205	959277	273300	125548	68458	89273

Table C.21: *TSP(9) – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	40284900	9170942	2153612	547642	142919	42270	17978
P-Ideal	41708532	10508192	2695097	685780	184424	51549	23096
C-Ideal-1	41708550	10198072	2463512	635929	171185	52522	25641
C-Ideal-2	41708550	10248470	2457766	627366	167683	54446	26210
RR-1	41708532	9996082	2532657	667974	243249	141835	135856
RR-2	41708532	10047895	2547202	651861	222340	109408	113971
Diff-1	42453008	10753891	2828432	733163	301146	283301	– –
Diff-2	42453008	11890537	2876274	762911	246536	244190	– –
TTM	41708532	10359111	2493742	668676	217039	78791	48531
XTM	41708532	10288864	2489987	661523	218532	113345	78960
XTM-C	41708532	10510311	2785668	774362	293431	128218	158858

Table C.22: *TSP(10) – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	315040750	70025896	14156147	3044056	948044	197421	57253
P-Ideal	326034362	67583229	15875151	3913158	1009189	264798	85029
C-Ideal-1	326034380	60800979	14483211	3640721	954195	257512	88977
C-Ideal-2	326034380	60790603	14446087	3546869	923410	252880	87020
RR-1	326034362	60805849	14489584	3618116	1194730	608160	477571
RR-2	326034362	60805439	14484518	3595915	1006446	415116	325957
Diff-1	331848858	80648967	15999477	4115066	1102629	798231	– –
Diff-2	331848858	82344330	16588289	4588820	1143817	670868	– –
TTM	326034362	75165863	17027508	4171091	1041265	332591	143688
XTM	326034362	60856918	14520085	3841879	1089472	450306	247843
XTM-C	326034362	75548463	19239072	4262388	1450355	555329	377919

Table C.23: *TSP(11) – Running Times (cycles).*

C.3.2 Variable t_n

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	40284900	9176567	2153693	542931	142919	43319	19411
P-Ideal	41708532	10818194	2823914	686782	177128	53554	22804
C-Ideal-1	37268656	8938208	2277964	582686	157152	51602	27831
C-Ideal-2	37268656	9183199	2325136	567449	154608	52818	27902
RR-1	41708532	10277961	2607727	674641	269242	179227	185395
RR-2	41708532	10286993	2608425	663625	258329	156383	149001
Diff-1	42452926	11835760	3068563	794801	328435	307961	– –
Diff-2	42452926	12295460	3248999	841093	326841	330127	– –
TTM	41708532	10420442	2597085	702539	218844	91096	46915
XTM	41708532	10210999	2522599	665090	257221	139872	79813
XTM-C	41708532	10523375	2799160	782442	299168	141508	128238

Table C.24: *TSP(10) – t_n = 2 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	40284926	9030180	2154774	541963	143003	41477	19069
P-Ideal	37268674	9753003	2602473	623167	164079	49003	24135
C-Ideal-1	37268728	8939314	2312564	583627	159735	54946	28877
C-Ideal-2	37268728	9176764	2329364	568503	155500	53938	29785
RR-1	37268680	9206744	2221696	675244	280451	226709	214749
RR-2	37268680	9218272	2358436	659935	283511	200450	186491
Diff-1	38004678	12072635	3174203	852180	383371	377469	– –
Diff-2	38004678	12072548	3270547	870392	345094	338349	– –
TTM	37268756	9203473	2454967	660813	203593	91824	51894
XTM	37268756	9235892	2236277	635671	246553	117606	75326
XTM-C	37268740	9375229	2538273	717532	284910	128558	189253

Table C.25: *TSP(10) – t_n = 4 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	40284978	9211855	2244573	545730	142692	44985	20207
P-Ideal	37268782	9825104	2622276	629292	166593	50832	25689
C-Ideal-1	37268836	8940971	2254033	581546	170192	58026	33841
C-Ideal-2	37268836	9515668	2337640	571036	170004	57057	33969
RR-1	37268784	9194751	2358986	710430	338604	326150	312106
RR-2	37268784	9213020	2326764	733383	346484	334363	257655
Diff-1	37933850	14963301	4036970	1109184	507669	535733	– –
Diff-2	37933850	14795072	4143477	1191087	480872	543859	– –
TTM	37268918	9219140	2372481	662891	214460	95477	73486
XTM	37268918	9224555	2349966	616615	220696	96298	82983
XTM-C	37268830	9365410	2547801	763238	255896	126461	120770

Table C.26: *TSP(10) – t_n = 8 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	38269172	8956924	2230310	543726	143442	43646	21654
P-Ideal	36262376	9675295	2463593	622066	164949	51854	28741
C-Ideal-1	36262430	8941216	2264960	594370	168014	61959	41406
C-Ideal-2	36262430	9169046	2288068	578868	170942	64228	42374
RR-1	36262370	8953442	2298015	708163	433603	387426	401201
RR-2	36262370	9196652	2382659	823730	501444	434040	454685
Diff-1	36989341	18241108	5591049	1601684	879497	794900	– –
Diff-2	36989341	19612318	6200907	1773702	851644	885747	– –
TTM	36262602	9038395	2313249	697930	287532	128701	102346
XTM	36262602	9288592	2383363	627707	236523	123597	87116
XTM-C	36262442	9143246	2495478	773513	285475	141128	117284

Table C.27: *TSP(10) – t_n = 16 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	35747304	8661550	2157590	518554	146707	49450	27169
P-Ideal	35125384	9015423	2449341	642126	174718	67864	48381
C-Ideal-1	35139827	8438390	2233736	609085	192904	101322	89819
C-Ideal-2	35139827	8743088	2238134	619288	204183	120543	191787
RR-1	35125382	8802746	2393383	987733	693296	679959	673072
RR-2	35125382	9071328	3103631	1525561	1410619	1693728	983679
Diff-1	35712861	44713664	16255811	4879927	2499625	2493885	– –
Diff-2	35712861	46952858	15475347	4958644	2593371	2733879	– –
TTM	35126186	9088347	2505409	866710	423920	260616	336035
XTM	35126186	9027484	2530795	789966	324328	262660	225875
XTM-C	35125796	9169307	2645049	881426	451001	378891	386189

Table C.28: *TSP(10) – t_n = 64 Cycles / Flit-Hop – Running Times (cycles).*

C.4 UNBAL

C.4.1 $t_n = 1$ Cycle / Flit-Hop

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	601423	150552	37856	9712	2794	1193	1061	1041
Stat	601451	151116	38503	10513	3837	2745	– –	– –
P-Ideal	601451	172279	43975	11901	11143	11143	– –	– –
C-Ideal-1	601469	252194	83453	48228	48236	48179	– –	– –
C-Ideal-2	601469	150932	39699	11790	4659	3032	3215	3418
RR-1	601451	219990	81157	48730	52629	60438	– –	– –
RR-2	601451	151408	41001	13621	9918	9964	9964	– –
Diff-1	612669	165953	49035	42096	41461	41414	– –	– –
Diff-2	612669	174799	48831	38377	38381	38381	– –	– –
TTM	601451	153845	41067	13657	7987	7969	7567	7978
XTM	601451	152570	45179	15052	8778	7120	7402	6769
XTM-C	601451	153482	42785	18586	12128	12555	12853	13973

Table C.29: *UNBAL(1024) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	2404687	601368	150560	37888	9838	2954	1337	1213
Stat	2404715	601932	151207	38689	10881	4573	4243	– –
P-Ideal	2404715	688383	173725	44712	41863	– –	– –	– –
C-Ideal-1	2404733	1005902	331239	189530	189540	– –	– –	– –
C-Ideal-2	2404733	601748	153127	41090	12241	5135	3420	4158
RR-1	2404715	876055	322285	190020	193884	– –	– –	– –
RR-2	2404715	602224	155512	44082	19137	19393	19327	– –
Diff-1	2448707	655806	175443	82893	82891	83576	– –	– –
Diff-2	2448707	691825	183335	77300	79073	78355	– –	– –
TTM	2404715	604955	154410	44036	17450	10850	10214	10680
XTM	2404715	603999	168387	50501	23673	12772	11159	10280
XTM-C	2404715	604513	158667	54386	30082	19281	17258	17765

Table C.30: *UNBAL(4096) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	9617743	2404632	601376	150592	38014	9998	3098	1466
Stat	9617771	2405196	602023	151393	39057	11617	6045	7343
P-Ideal	9617771	2752767	693743	175652	164761	– –	– –	– –
C-Ideal-1	9617789	4021080	1321857	754685	754703	– –	– –	– –
C-Ideal-2	9617789	2405012	604920	156373	42418	12682	5666	5202
RR-1	9617771	3504936	1287644	755413	759764	– –	– –	– –
RR-2	9617771	2405488	607116	159412	50449	45451	45390	– –
Diff-1	9790833	2613558	680367	194927	165819	165819	– –	– –
Diff-2	9790833	2757652	722464	192409	158327	158215	– –	– –
TTM	9617771	2408675	605047	156720	44180	18919	13928	14276
XTM	9617771	2407272	653619	176066	60498	32323	17090	14391
XTM-C	9617771	2408467	611832	203270	77201	40920	28618	26859

Table C.31: *UNBAL(16384) – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	38469967	9617688	2404640	601408	150718	38174	10142	3227
Stat	38469995	9618252	2405287	602209	151761	39793	13089	9265
P-Ideal	38469995	11010295	2772303	699430	656281	– –	– –	– –
C-Ideal-1	38470013	16082583	5284667	3015695	3015695	– –	– –	– –
C-Ideal-2	38470013	9618068	2409301	609976	160415	43098	13519	6791
RR-1	38469995	14013649	5144331	3016122	3020315	– –	– –	– –
RR-2	38469995	9618544	2413725	616066	172794	113002	110970	– –
Diff-1	39159316	10443171	2699916	705746	326855	327737	– –	– –
Diff-2	39159316	10836040	2881729	745833	315577	317237	– –	– –
TTM	38469995	9622751	2409216	611494	161329	48019	22966	16276
XTM	38469995	9620153	2581413	663652	220112	86345	31563	23082
XTM-C	38469995	9621468	2420705	679842	225614	106160	51351	42882

Table C.32: *UNBAL(65536) – Running Times (cycles).*

C.4.2 Variable t_n

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	9617743	2404639	601377	150587	38047	10027	3101	1430
Stat	9617771	2405193	602025	151393	39057	11617	6347	7628
P-Ideal	9617771	2752791	693761	175705	164779	– –	– –	– –
C-Ideal-1	9617789	4071548	1361492	754675	754783	– –	– –	– –
C-Ideal-2	9617789	2405040	606718	160359	42102	13424	6439	6848
RR-1	9617771	3553078	1448347	806942	757143	– –	– –	– –
RR-2	9617771	2405541	607736	157020	50633	50529	51265	– –
Diff-1	9790833	2776406	741372	215166	184825	184825	– –	– –
Diff-2	9790833	2929302	779836	212838	176715	177217	– –	– –
TTM	9617771	2409171	611465	158732	48466	19116	15863	15170
XTM	9617771	2406975	612818	176934	60980	30603	18385	14383
XTM-C	9617771	2408469	618841	183654	72119	36257	30331	30669

Table C.33: *UNBAL(16384) – t_n = 2 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	9617769	2404671	601403	150587	37997	9935	2991	1330
Stat	9617797	2405223	602051	151419	39203	11997	6919	8746
P-Ideal	9617807	2752863	693851	175741	164887	– –	– –	– –
C-Ideal-1	9617825	4170649	1421874	754783	754847	– –	– –	– –
C-Ideal-2	9617825	2405126	607960	159723	44227	14748	8883	11462
RR-1	9617797	3924496	1820303	1092005	814483	– –	– –	– –
RR-2	9617797	2406452	607471	160118	65828	66643	67182	– –
Diff-1	9790839	3101269	862372	255089	221957	221973	– –	– –
Diff-2	9790839	3207796	897689	253329	212636	212614	– –	– –
TTM	9617889	2405702	611702	156291	48826	23947	18998	21534
XTM	9617899	2407064	612264	171181	70299	30271	19512	16630
XTM-C	9617817	2410994	621515	194880	66717	38731	34593	30769

Table C.34: *UNBAL(16384) – t_n = 4 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)							
Mgr.	1	4	16	64	256	1024	4096	16384
Free-Ideal	9617821	2404715	601447	150631	37935	9873	2927	1266
Stat	9617875	2405299	602171	151669	39593	12673	8115	11008
P-Ideal	9617861	2752969	694029	175959	165049	– –	– –	– –
C-Ideal-1	9617897	4365593	1548553	754959	755143	– –	– –	– –
C-Ideal-2	9617897	2405416	606438	159604	45425	17084	12060	17379
RR-1	9617875	4514350	2436622	1616026	1256941	– –	– –	– –
RR-2	9617875	2406931	607667	162251	85907	84961	83197	– –
Diff-1	9790841	3752572	1101869	334303	293799	293830	– –	– –
Diff-2	9790841	3848858	1136623	331840	281839	282227	– –	– –
TTM	9617997	2412800	609717	162150	57617	27659	27477	26717
XTM	9618071	2407142	614168	169452	70525	39641	30924	25241
XTM-C	9617953	2410835	626021	192675	76777	31209	30432	35682

Table C.35: *UNBAL(16384) – t_n = 8 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	9617925	2404825	601555	150735	38031	9871	2927
Stat	9618031	2405679	602613	152267	40425	13999	8447
P-Ideal	9618023	2753261	694249	176581	165427	– –	– –
C-Ideal-1	9618059	4723315	1791979	755411	755377	– –	– –
C-Ideal-2	9618059	2405902	608150	163851	52190	21361	20516
RR-1	9618031	5386174	3382293	2456214	2004639	– –	– –
RR-2	9618031	2408169	612522	175209	113314	113940	113940
Diff-1	9790835	5056912	1578660	491322	436759	434702	– –
Diff-2	9790835	5177838	1615852	487891	418464	417133	– –
TTM	9618249	2413018	612189	172037	59377	42768	36099
XTM	9618139	2410285	627809	193619	81576	39756	37971
XTM-C	9618139	2410051	629739	193416	66044	44412	39689

Table C.36: *UNBAL(16384) – t_n = 16 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)						
Mgr.	1	4	16	64	256	1024	4096
Free-Ideal	9618497	2405385	602123	151307	38603	10427	3383
Stat	9618993	2407995	605315	155699	45443	22059	23033
P-Ideal	9618995	2754645	696277	179861	168321	– –	– –
C-Ideal-1	9619031	6141369	2867639	1086767	759677	– –	– –
C-Ideal-2	9619031	2409300	620783	187299	71467	50757	74059
RR-1	9618993	7516640	6079406	5169356	4650767	– –	– –
RR-2	9618993	2418036	653437	233753	218988	219018	219048
Diff-1	9791855	12874161	4393430	1433882	1273437	1273225	– –
Diff-2	9791855	13157159	4471812	1418808	1225454	1231163	– –
TTM	9619797	2430299	657701	215213	127622	118838	122469
XTM	9619511	2430879	644648	234714	124675	85494	90431
XTM-C	9619511	2426418	640117	219719	107551	89765	95154

Table C.37: *UNBAL(16384) – t_n = 64 Cycles / Flit-Hop – Running Times (cycles).*

C.5 MATMUL: Coarse, Cached

C.5.1 $t_n = 1$ Cycle / Flit-Hop

	p (number of processors)			
Mgr.	1	4	16	64
Free-Ideal	202748	85405	30022	16537
Stat	202748	73246	26632	14552
P-Ideal	202748	155980	52801	26328
C-Ideal-1	202766	85546	32618	29992
C-Ideal-2	202766	85420	31749	30742
RR-1	202748	86639	31723	23556
RR-2	202748	86639	31731	23916
Diff-1	207312	305232	106888	92857
Diff-2	207312	94855	66531	56341
TTM	202748	86551	31932	21687
XTM	202748	86311	32470	22454

Table C.38: *MATMUL(16) (coarse, cached) – Running Times (cycles).*

	p (number of processors)				
Mgr.	1	4	16	64	256
Free-Ideal	1596780	522831	151081	55333	33011
Stat	1596780	475054	139342	49758	29680
P-Ideal	1596780	996878	285914	99288	52622
C-Ideal-1	1596798	519578	154498	106256	38648
C-Ideal-2	1596798	519452	153179	110077	40071
RR-1	1596780	524139	153245	62641	62336
RR-2	1596780	524139	153371	61945	67559
Diff-1	1625990	2042920	609992	383851	324903
Diff-2	1625990	571057	315347	196457	144490
TTM	1596780	524581	153364	59810	42205
XTM	1596780	523784	153964	61025	46732

Table C.39: *MATMUL(32) (coarse, cached) – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	12677708	3642945	975307	293114	115611	78707
Stat	12677708	3454414	930719	272724	101960	68808
P-Ideal	12677708	7103884	1900267	559296	212970	127776
C-Ideal-1	12677726	3626061	981311	574234	123416	91185
C-Ideal-2	12677726	3626236	978331	587114	127164	94167
RR-1	12677708	3644334	979545	301534	154241	189835
RR-2	12677708	3644334	980529	301534	178596	189722
Diff-1	12905516	14956668	4150412	2410011	1690097	1434242
Diff-2	12905516	3962996	2103620	983793	527595	490766
TTM	12677708	3647009	979384	301944	129253	98591
XTM	12677708	3643966	980768	302552	132229	109426

Table C.40: *MATMUL(64) (coarse, cached) – Running Times (cycles).*

C.5.2 Variable t_n

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	12677708	3957887	1114522	359008	171385	143073
Stat	12677708	3662215	1025773	323144	137551	109961
P-Ideal	12677708	7675991	2124355	678152	300156	210318
C-Ideal-1	12677726	4003137	1109886	653285	183618	163289
C-Ideal-2	12677726	4003209	1119695	721482	182743	161251
RR-1	12677708	3971817	1111480	378743	222136	296526
RR-2	12677708	3971817	1107161	381229	279793	309123
Diff-1	12905516	16812535	4983120	3135747	2308575	2303854
Diff-2	12905516	4617377	2511591	1194111	798020	1038068
TTM	12677708	4003598	1118800	380311	183951	160132
XTM	12677708	4003087	1115395	386395	195183	173381

Table C.41: *MATMUL(64) (coarse, cached) –* $t_n = 2$ *Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	12677734	4671729	1361143	500790	269279	245655
Stat	12677734	4077887	1212061	423526	209041	192405
P-Ideal	12677744	8820271	2572751	1623274	821318	623514
C-Ideal-1	12677762	4663054	1393226	951210	298032	296650
C-Ideal-2	12677762	4721735	2552324	530808	303810	355427
RR-1	12677734	4733047	1375338	531811	364024	524772
RR-2	12677734	4733047	1373995	536321	477251	651379
Diff-1	12905522	20843250	6875081	4379999	3537360	3922007
Diff-2	12905522	6086145	3557104	1784404	1402990	1982218
TTM	12677826	4687756	1388807	531672	299655	280959
XTM	12677836	4662978	1390074	532596	317845	283898

Table C.42: *MATMUL(64) (coarse, cached) –* $t_n = 4$ *Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	12677786	5976757	1885091	767245	473356	489364
Stat	12677786	4909247	1573680	622191	352013	357029
P-Ideal	12677780	19367269	6617909	4626254	2491020	2045932
C-Ideal-1	12677798	6045105	1932253	836134	527621	542246
C-Ideal-2	12677798	5961410	1918050	1401302	802316	544580
RR-1	12677786	6181500	1903260	839083	673275	1076066
RR-2	12677786	6181500	1918744	852106	816666	1197341
Diff-1	12905508	30206673	11568874	8175276	6822981	10101494
Diff-2	12905508	9620614	6085170	2803742	2844984	4614953
TTM	12677916	6093536	1918809	838308	529080	543875
XTM	12677926	6159988	1931450	838568	529658	575220

Table C.43: *MATMUL(64) (coarse, cached) –* $t_n = 8$ *Cycles / Flit-Hop – Running Times (cycles).*

C.6 MATMUL: Fine, Cached

C.6.1 $t_n = 1$ Cycle / Flit-Hop

	p (number of processors)			
Mgr.	1	4	16	64
Free-Ideal	206588	98109	41702	30878
Stat	206588	75586	28785	18080
P-Ideal	206588	87531	50217	28605
C-Ideal-1	206606	87737	42438	30552
C-Ideal-2	206606	87614	46266	30994
RR-1	206588	94881	42822	32049
RR-2	206588	94881	42568	35718
Diff-1	211181	159275	70891	62538
Diff-2	211181	106361	59980	46141
TTM	206588	97739	45650	36879
XTM	206588	97495	47762	37482

Table C.44: *MATMUL(16) (fine, cached) – Running Times (cycles).*

	p (number of processors)				
Mgr.	1	4	16	64	256
Free-Ideal	1612140	564232	190014	91165	70210
Stat	1612140	483180	144385	54702	36922
P-Ideal	1612140	530486	244005	95223	51175
C-Ideal-1	1612158	527529	194412	81280	58083
C-Ideal-2	1612158	527406	207535	83619	63370
RR-1	1612140	554513	192097	87192	73708
RR-2	1612140	554513	193783	91924	90078
Diff-1	1641775	1039193	408126	201852	161551
Diff-2	1641775	618588	276325	121551	108735
TTM	1612140	566269	196513	102050	75651
XTM	1612140	566021	199196	97434	86918

Table C.45: *MATMUL(32) (fine, cached) – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	12739148	3841845	1120468	408020	215175	188504
Stat	12739148	3485580	947240	283918	112888	83668
P-Ideal	12739148	3673909	1532939	437126	171883	125121
C-Ideal-1	12739166	3657353	1136724	369822	170071	136810
C-Ideal-2	12739166	3657230	1275774	388444	185461	143820
RR-1	12739148	3763194	1131171	389605	193748	232248
RR-2	12739148	3763194	1130461	403760	233886	263437
Diff-1	12968121	7534645	2246912	1101161	515466	577223
Diff-2	12968121	4250445	1498754	566448	278983	338331
TTM	12739148	3810409	1157229	412023	209896	192461
XTM	12739148	3810157	1144804	406853	214161	211593

Table C.46: *MATMUL(64) (fine, cached) – Running Times (cycles).*

C.6.2 Variable t_n

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	12739148	4230706	1461846	569716	344047	325631
Stat	12739148	3693069	1042397	334749	149421	124975
P-Ideal	12739148	4033242	1802870	599083	251095	195476
C-Ideal-1	12739166	4033821	1391797	500962	243240	203183
C-Ideal-2	12739166	4033922	1399579	513653	255174	236947
RR-1	12739148	4291014	1376186	499651	284547	385371
RR-2	12739148	4189308	1417673	582338	399629	480874
Diff-1	12968121	8455841	2595856	1478620	814831	911810
Diff-2	12968121	5152450	1959171	819888	488674	434751
TTM	12739148	4243316	1469110	613282	324196	292144
XTM	12739148	4224525	1710448	597103	334786	396301

Table C.47: *MATMUL(64) (fine, cached) – t_n = 2 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	12739174	5479655	1939592	919830	595148	582113
Stat	12739174	4108928	1229619	436145	222657	207695
P-Ideal	12739184	4751970	2523022	822453	432767	345164
C-Ideal-1	12739202	4694431	1875179	737853	407837	367605
C-Ideal-2	12739202	5107929	1899364	824117	474256	460087
RR-1	12739174	5034938	1763291	738044	461916	679977
RR-2	12739174	5105081	1887344	912465	716178	1034937
Diff-1	12968147	10456151	3949960	1964292	1118369	1711399
Diff-2	12968147	7235875	2945461	1499787	893749	833174
TTM	12739266	5145965	2007646	938104	523968	502553
XTM	12739276	5034013	1936568	904656	571756	594734

Table C.48: *MATMUL(64) (fine, cached) – t_n = 4 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	12739226	7244817	2822331	1573203	1097566	1124149
Stat	12739226	4940097	1592199	636931	368665	373035
P-Ideal	12739220	11276943	3474505	844856	942091	679288
C-Ideal-1	12739238	6076809	2748770	1184658	711988	674017
C-Ideal-2	12739238	6731273	2811055	1413270	920755	955788
RR-1	12739226	6733202	2541415	1089971	805437	1234849
RR-2	12739226	6697210	3046785	1529806	1432862	2474518
Diff-1	12968103	15091855	6980501	4039756	2802365	3605615
Diff-2	12968103	10664163	5885315	3157172	2189743	2089678
TTM	12739356	6951140	2942752	1491604	953380	965724
XTM	12739366	6876345	3394139	1520421	980527	1002713

Table C.49: *MATMUL(64) (fine, cached) – t_n = 8 Cycles / Flit-Hop – Running Times (cycles).*

C.7 MATMUL: Coarse, Uncached

C.7.1 $t_n = 1$ Cycle / Flit-Hop

	p (number of processors)			
Mgr.	1	4	16	64
Free-Ideal	352508	240678	79771	34179
Stat	352508	167642	59880	27422
P-Ideal	352508	468542	157456	65735
C-Ideal-1	352526	256062	136715	60244
C-Ideal-2	352526	256080	136429	61813
RR-1	352508	256350	83284	39200
RR-2	352508	256350	82216	39955
Diff-1	359550	920988	329679	267911
Diff-2	359550	278965	167711	122882
TTM	352508	255493	83489	40247
XTM	352508	255274	83843	43488

Table C.50: *MATMUL(16) (coarse, uncached) – Running Times (cycles).*

	p (number of processors)				
Mgr.	1	4	16	64	256
Free-Ideal	2719340	1854235	545222	177033	86106
Stat	2719340	1265849	416058	136358	64582
P-Ideal	2719340	3518236	1111024	374400	165215
C-Ideal-1	2719358	1911692	941978	303277	124840
C-Ideal-2	2719358	1911692	941700	318808	137320
RR-1	2719340	1911665	563090	191270	119261
RR-2	2719340	1911665	562283	191270	148610
Diff-1	2768358	6937270	2332532	1539427	1554854
Diff-2	2768358	2075989	1109216	396603	346164
TTM	2719340	1911289	563301	190608	95920
XTM	2719340	1909555	562160	194963	103078

Table C.51: *MATMUL(32) (coarse, uncached) – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	21361484	14616058	4098472	1252586	465082	251516
Stat	21361484	9901976	3189039	966169	324456	171770
P-Ideal	21361484	27300058	8394160	2662464	993452	475717
C-Ideal-1	21361502	14842476	7144121	2086614	699671	292729
C-Ideal-2	21361502	14842476	7143944	2210214	737067	284211
RR-1	21361484	14842428	4234827	1343622	522189	400868
RR-2	21361484	14842428	4234827	1343622	514305	424984
Diff-1	21745016	53908520	17628461	11008495	8519769	– –
Diff-2	21745016	16109183	8286869	2925353	1518827	– –
TTM	21361484	14842060	4234487	1342960	502354	262824
XTM	21361484	14837264	4234703	1343355	530603	301307

Table C.52: *MATMUL(64) (coarse, uncached) – Running Times (cycles).*

C.7.2 Variable t_n

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	21525444	24315676	6772491	2116706	803574	506830
Stat	21525444	14445449	5007631	1571689	545821	304603
P-Ideal	21525444	43885347	14049797	4607224	1786114	881951
C-Ideal-1	21525462	24325948	7001378	2319178	1146674	524898
C-Ideal-2	21525462	24326002	12410385	2326264	1247303	547914
RR-1	21525444	24326392	7099113	2326224	903004	760273
RR-2	21525444	24326392	7099113	2341043	913442	781196
Diff-1	21911988	88868137	31243325	20490675	14961017	– –
Diff-2	21911988	28054956	15505433	4992768	2868938	– –
TTM	21525444	24315762	7114661	2325380	901946	560805
XTM	21525444	24318261	7098634	2325622	902170	505961

Table C.53: *MATMUL(64) (coarse, uncached) – t_n = 2 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	21853390	42734468	12086239	3833564	1529273	913449
Stat	21853390	23532545	8644873	2782885	988735	570531
P-Ideal	21853400	77056037	25361019	15568379	6644712	3346389
C-Ideal-1	21853418	43277393	12426237	6002250	1956558	1037355
C-Ideal-2	21853418	43292853	12827078	6326226	1714143	1001108
RR-1	21853390	43277928	12827177	4291610	1702392	1474251
RR-2	21853390	43277928	12858605	4291472	1710295	1473729
Diff-1	22245878	164351591	65073757	44780797	32965603	– –
Diff-2	22245878	43584348	29828811	10460863	7253197	– –
TTM	21853482	43273046	12826090	4290254	1701284	958138
XTM	21853492	43292710	12826406	4290516	1701680	1004857

Table C.54: *MATMUL(64) (coarse, uncached) – t_n = 4 Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	22509282	81192208	22817227	7416160	2955165	1834319
Stat	22509282	41706687	15919325	5205121	1874563	1102101
P-Ideal	22509276	247271123	91569791	61080065	25048328	12866223
C-Ideal-1	22509294	81194714	24282197	8220647	4417729	2019579
C-Ideal-2	22509294	81194822	24282150	8220403	3300310	1993246
RR-1	22509282	81189238	24283314	8222348	3324033	2829060
RR-2	22509282	81189238	24283222	8222256	3355445	3039472
Diff-1	22914102	385614536	157274311	107850359	62692673	– –
Diff-2	22914102	126621352	71501656	36834573	15225247	– –
TTM	22509412	81187665	24281540	8219678	3299472	1998814
XTM	22509422	81195257	24281956	8220220	3300050	2055491

Table C.55: *MATMUL(64) (coarse, uncached) – t_n = 8 Cycles / Flit-Hop – Running Times (cycles).*

C.8 MATMUL: Fine, Uncached

C.8.1 $t_n = 1$ Cycle / Flit-Hop

	p (number of processors)			
Mgr.	1	4	16	64
Free-Ideal	370684	250140	87530	41128
Stat	370684	177405	65854	33010
P-Ideal	370684	269771	122941	43855
C-Ideal-1	370702	270194	93391	46055
C-Ideal-2	370702	270194	93190	46376
RR-1	370684	235411	92211	47934
RR-2	370684	235411	91800	51745
Diff-1	377815	442983	194642	129980
Diff-2	377815	351660	121258	68191
TTM	370684	307706	93874	54221
XTM	370684	256891	94969	50456

Table C.56: *MATMUL(16) (fine, uncached) – Running Times (cycles).*

	p (number of processors)				
Mgr.	1	4	16	64	256
Free-Ideal	2792044	1890956	579679	192066	95283
Stat	2792044	1303664	437048	149906	78204
P-Ideal	2792044	1965900	837430	211729	101752
C-Ideal-1	2792062	1967841	591216	197795	102040
C-Ideal-2	2792062	1967841	582517	200821	99186
RR-1	2792044	1687823	586390	199798	116810
RR-2	2792044	1687823	586169	207368	129038
Diff-1	2842495	3225583	1275404	698435	322556
Diff-2	2842495	2176414	782205	280134	152494
TTM	2792044	1896675	581469	206778	112407
XTM	2792044	1896498	575366	206040	117651

Table C.57: *MATMUL(32) (fine, uncached) – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	21652300	14761832	4310075	1258299	454422	245181
Stat	21652300	10051914	3269131	1012307	360160	207622
P-Ideal	21652300	15057941	6120650	1385310	510794	248728
C-Ideal-1	21652318	15066041	4305572	1279048	457170	263063
C-Ideal-2	21652318	15066041	4275520	1302630	456183	255201
RR-1	21652300	12830193	4286631	1265566	490239	389978
RR-2	21652300	12830193	4277495	1308846	478205	351479
Diff-1	22042307	24687501	9370095	4527924	1925214	1490239
Diff-2	22042307	22131834	5515578	1771289	755502	381221
TTM	21652300	14771918	4205281	1325285	478531	295849
XTM	21652300	14648811	4183621	1386063	509640	293876

Table C.58: *MATMUL(64) (fine, uncached) – Running Times (cycles).*

C.8.2 Variable t_n

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	21980100	24690836	6839050	2160918	778212	428396
Stat	21980100	14681413	5134823	1645509	603049	359433
P-Ideal	21980100	24692899	10466311	2491841	873255	444641
C-Ideal-1	21980118	24709519	6995476	2202244	779288	439381
C-Ideal-2	21980118	24709494	7590497	2149679	788517	441756
RR-1	21980100	24709900	7033305	2179351	839849	744295
RR-2	21980100	19942596	7106609	2242132	846729	735381
Diff-1	22375593	39955831	17147603	7744693	3928718	2728089
Diff-2	22375593	32066133	10217894	3163537	1291874	665254
TTM	21980100	20236650	7030368	2216803	827828	485835
XTM	21980100	20236722	7113994	2227559	856148	482915

Table C.59: *MATMUL(64) (fine, uncached) – $t_n = 2$ Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	22635726	34523875	12826214	3907302	1450599	776006
Stat	22635726	23940537	8866307	2911965	1088959	663265
P-Ideal	22635736	43962980	18526642	5067859	1691956	850316
C-Ideal-1	22635754	43964034	12670352	3913197	1434321	820946
C-Ideal-2	22635754	43996199	13600879	4023379	1453164	824704
RR-1	22635726	43964713	12670904	3980386	1543615	1426384
RR-2	22635726	43964713	12738699	4037218	1656624	1606775
Diff-1	23043277	75314543	35094067	12392624	6762955	4680617
Diff-2	23043277	61019441	22382573	6749695	2731156	1470231
TTM	22635818	35050760	12489330	3983824	1595955	855229
XTM	22635828	34460805	12803942	4328527	1588434	862245

Table C.60: *MATMUL(64) (fine, uncached) – $t_n = 4$ Cycles / Flit-Hop – Running Times (cycles).*

	p (number of processors)					
Mgr.	1	4	16	64	256	1024
Free-Ideal	23946978	83584666	23243558	7312834	2729209	1490544
Stat	23946978	42458765	16329195	5444747	2060857	1270617
P-Ideal	23946972	145982731	49409278	8785571	3687700	1955760
C-Ideal-1	23946990	82568456	25688816	7530983	2688134	1546855
C-Ideal-2	23946990	63498041	24806013	7494771	2728884	1546171
RR-1	23946978	62516654	23005915	7560244	2969496	2801732
RR-2	23946978	82569647	24360516	7669920	3107173	2968417
Diff-1	24377597	165325301	110036655	32052696	14765763	11118528
Diff-2	24377597	133560749	45395985	16573886	7476504	3802137
TTM	23947108	64678466	23483204	8317440	2890945	1588285
XTM	23947118	64677713	24778887	8562822	3086391	1655296

Table C.61: *MATMUL(64) (fine, uncached) – $t_n = 8$ Cycles / Flit-Hop – Running Times (cycles).*

Bibliography

[1] Anant Agarwal. Overview of the Alewife Project. Alewife Systems Memo #10., July 1990.

[2] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[3] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computing: Numerical Methods.* Prentice Hall, 1989.

[4] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[5] William J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, C-39(6):775–785, June 1990.

[6] Alvin M. Despain and David A. Patterson. X-TREE: A Tree Structured Multi-Processor Computer Architecture. In *Proceedings of the Fifth International Symposium on Computer Architecture*, 1978.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[8] Robert H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[9] Robert H. Halstead and Stephen A. Ward. The MuNet: A Scalable Decentralized Architecture for Parallel Computation. In *Proceedings of the Seventh International Symposium on Computer Architecture*, 1980.

[10] Kirk Johnson. The Impact of Communication Locality on Large-Scale Multiprocessor Performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[11] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *Proceedings of Practice and Principles of Parallel Programming (PPoPP) 1993*, 1993.

[12] David A. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implementation*, 1989.

[13] Orly Kremien and Jeff Kramer. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):747–760, November 1992.

[14] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, October 1985.

[15] John Kubiatowicz. User's Manual for the A-1000 Communications and Memory Management Unit. Alewife Memo No. 19, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

[16] Vipin Kumar, Grama Y. Ananth, and Vempaty Nageshwara Rao. Scalable Load Balancing Techniques for Parallel Computers. Technical Report 92-021, Army High Performance Computing Research Center, University of Minnesota, January 1992.

[17] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[18] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem.* John Wiley and Sons, 1985.

[19] Frank C. H. Lin and Robert M. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, SE-13(1):32–38, January 1987.

[20] Lionel M. Ni, Chong-Wei Xu, and Thomas B. Gendreau. A Distributed Drafting Algorithm for Load Balancing. *IEEE Transactions on Software Engineering*, SE-11(10):1153–1161, October 1985.

[21] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[22] William H. Press, Brian P. Flannery, Saul A Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[23] J. Rees and N. Adams. T: A Dialect of LISP. In *Proceedings of Symposium on Lisp and Functional Programming*, 1982.

[24] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proceedings of the Third Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 1991.

[25] C. H. Sequin, A. M. Despain, and D. A. Patterson. Communication in X-TREE: A Modular Multiprocessor System. In *Proceedings of the 1978 Annual Conference of the Association for Computing Machinery*, 1978.

[26] H. Sullivan and T. R. Bashkow. A Large Scale, Homogeneous, Fully Distributed Parallel Machine. In *Proceedings of the Fourth Annual Symposium on Computer Architecture*, 1977.

[27] Stephen A. Ward and Jr. Robert H. Halstead. *Computation Structures*. MIT Press, 1990.

[28] Min-You Wu and Wei Shu. Scatter Scheduling for Problems with Unpredictable Structures. In *Sixth Distributed Memory Conference Proceedings*, 1991.

[29] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[30] Songnian Zhou and Timothy Brecht. Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1991.

[31] Taieb F. Znati, Rami G. Melhem, and Kirk R. Pruhs. Dilation Based Bidding Schemes for Dynamic Load Balancing on Distributed Processing Systems. In *Sixth Distributed Memory Conference Proceedings*, 1991.