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

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.

---

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

1 Introduction 11
   1.1 Principles for Algorithm Design in Distributed Environments 12
   1.2 Contributions of This Thesis 13
   1.3 Systems Framework and Assumptions 16
   1.4 Terminology and Notation 19
   1.5 Outline 19

2 Background 20
   2.1 Centralized Self-Scheduling 21
   2.2 Fully Distributed On-Line Algorithms 22
      2.2.1 Bidding Methods 22
      2.2.2 Drafting Methods 24
      2.2.3 Hybrid Methods 24

3 High-Level System Design 27
   3.1 Global Information Management 28
      3.1.1 The Need for Global Information 29
      3.1.2 Prospective Solutions 30
   3.2 Matching Threads with Processors 32
      3.2.1 X-Tree-Based Design 33
   3.3 Shared-Memory vs. Message-Passing 34

4 X-Tree-Based Thread Management 36
   4.1 X-Trees 39
      4.1.1 Notation 39
      4.1.2 Embedding a Tree in a $k$-ary $n$-Dimensional Mesh 42
   4.2 Updating Information in the X-Tree 47
   4.3 Thread Search 49
   4.4 An Example 50

5 Analysis 60
   5.1 One-Dimensional Case 62
      5.1.1 Search 62
      5.1.2 Presence Bit Update 67
   5.2 Two-Dimensional Case 72
# List of Tables

1.1 Machine Parameter Notation ................................................. 19
6.1 Timing Parameters .......................................................... 90
6.2 Application Characteristics ................................................ 101
7.1 Experiments Performed ($t_n = 1$) ...................................... 109
7.2 Experiments Performed (variable $t_n$) ................................ 110
7.3 Regions of Interest .......................................................... 114

C.1 AQ(0.5) – Running Times .................................................. 184
C.2 AQ(0.1) – Running Times .................................................. 185
C.3 AQ(0.05) – Running Times .................................................. 185
C.4 AQ(0.01) – Running Times .................................................. 185
C.5 AQ(0.005) – Running Times ................................................ 186
C.6 AQ(0.001) – Running Times ................................................ 186
C.7 AQ(0.01) – $t_n = 2$ Cycles / Flit-Hop – Running Times ............... 187
C.8 AQ(0.01) – $t_n = 4$ Cycles / Flit-Hop – Running Times ............... 187
C.9 AQ(0.01) – $t_n = 8$ Cycles / Flit-Hop – Running Times ............... 187
C.10 AQ(0.01) – $t_n = 16$ Cycles / Flit-Hop – Running Times ............. 188
C.11 AQ(0.01) – $t_n = 64$ Cycles / Flit-Hop – Running Times ............. 188
C.12 FIB(15) – Running Times .................................................. 189
C.13 FIB(20) – Running Times .................................................. 189
C.14 FIB(25) – Running Times .................................................. 190
C.15 FIB(20) – $t_n = 2$ Cycles / Flit-Hop – Running Times ............... 191
C.16 FIB(20) – $t_n = 4$ Cycles / Flit-Hop – Running Times ............... 191
C.17 FIB(20) – $t_n = 8$ Cycles / Flit-Hop – Running Times ............... 191
C.18 FIB(20) – $t_n = 16$ Cycles / Flit-Hop – Running Times ............. 192
C.19 FIB(20) – $t_n = 64$ Cycles / Flit-Hop – Running Times ............. 192
C.20 TSP(8) – Running Times .................................................... 193
C.21 TSP(9) – Running Times .................................................... 193
C.22 TSP(10) – Running Times .................................................. 194
C.23 TSP(11) – Running Times .................................................. 194
C.24 TSP(10) – $t_n = 2$ Cycles / Flit-Hop – Running Times ............... 195
C.25 TSP(10) – $t_n = 4$ Cycles / Flit-Hop – Running Times ............... 195
C.26 TSP(10) – $t_n = 8$ Cycles / Flit-Hop – Running Times ............... 195
C.27 TSP(10) – $t_n = 16$ Cycles / Flit-Hop – Running Times ............. 196
C.28 TSP(10) - \( t_n = 64 \) Cycles / Flit-Hop - Running Times
C.29 UNBAL(1024) - Running Times
C.30 UNBAL(4096) - Running Times
C.31 UNBAL(16384) - Running Times
C.32 UNBAL(65536) - Running Times
C.33 UNBAL(16384) - \( t_n = 2 \) Cycles / Flit-Hop - Running Times
C.34 UNBAL(16384) - \( t_n = 4 \) Cycles / Flit-Hop - Running Times
C.35 UNBAL(16384) - \( t_n = 8 \) Cycles / Flit-Hop - Running Times
C.36 UNBAL(16384) - \( t_n = 16 \) Cycles / Flit-Hop - Running Times
C.37 UNBAL(16384) - \( t_n = 64 \) Cycles / Flit-Hop - Running Times
C.38 MATMUL(16)
C.39 MATMUL(32)
C.40 MATMUL(64)
C.41 MATMUL(64) - \( t_n = 2 \) Cycles / Flit-Hop - Running Times
C.42 MATMUL(64) - \( t_n = 4 \) Cycles / Flit-Hop - Running Times
C.43 MATMUL(64) - \( t_n = 8 \) Cycles / Flit-Hop - Running Times
C.44 MATMUL(16) - \( t_n = 2 \) Cycles / Flit-Hop - Running Times
C.45 MATMUL(32) - \( t_n = 4 \) Cycles / Flit-Hop - Running Times
C.46 MATMUL(64) - \( t_n = 8 \) Cycles / Flit-Hop - Running Times
C.47 MATMUL(64) - \( t_n = 16 \) Cycles / Flit-Hop - Running Times
C.48 MATMUL(64) - \( t_n = 64 \) Cycles / Flit-Hop - Running Times
# List of Figures

1-1 Self-Scheduled Model ............................................. 17  
1-2 Self-Scheduled Implementation ................................. 18  
2-1 Taxonomy of Load Sharing Methods ............................. 21  
3-1 Combining Tree with Exact Queue Lengths .................. 31  
3-2 Combining Tree with Presence Bits ............................ 32  
4-1 Binary X-Tree on a One-Dimensional Mesh Network ......... 37  
4-2 Quad-X-Tree on a Two-Dimensional Mesh Network .......... 38  
4-3 Embedding a Binary Tree in a One-Dimensional Mesh: Naive Solution ..................................................... 42  
4-4 Embedding a Binary Tree in a One-Dimensional Mesh: Better Solution .................................................... 43  
4-5 Embedding a Quad-Tree in a Two-Dimensional Mesh ....... 45  
4-6 Two-Dimensional Presence Bit Cache Update ............... 49  
4-7 Thread Search Example – I ....................................... 51  
4-8 Thread Search Example – II ...................................... 52  
4-9 Thread Search Example – III ..................................... 53  
4-10 Thread Search Example – IV .................................... 54  
4-11 Thread Search Example – V ..................................... 55  
4-12 Thread Search Example – VI .................................... 56  
4-13 Thread Search Example – VII ................................... 57  
4-14 Thread Search Example – VIII .................................. 58  
4-15 Thread Search Example – IX .................................... 59  
5-1 One-Dimensional Search – Worst Case ......................... 64  
5-2 Two-Dimensional Search – Worst Case ......................... 74  
6-1 The Alewife Machine .............................................. 86  
6-2 NWO Simulator Organization ...................................... 87  
6-3 PISCES Multithreader Organization ............................ 87  
6-4 A Typical PISCES Thread ......................................... 88  
6-5 PISCES Thread Execution Order ................................. 88  
6-6 PISCES Alewife Simulation ....................................... 89  
6-7 Thread Queue Management ....................................... 97  
6-8 Coarse-Grained MATMUL Partitioning ....................... 99  
6-9 Fine-Grained MATMUL Partitioning ............................ 100
7-1 Experimental Parameters ............................................. 105
7-2 AQ: Regions of Interest ................................................. 112
7-3 FIB: Regions of Interest ................................................. 113
7-4 TSP: Regions of Interest ................................................. 113
7-5 UNBAL: Regions of Interest ........................................... 113
7-6 AQ: XTM vs. P-Ideal, C-Ideal-1 and C-Ideal-2 ................. 115
7-7 FIB: XTM vs. P-Ideal, C-Ideal-1 and C-Ideal-2 ................. 115
7-8 TSP: XTM vs. P-Ideal, C-Ideal-1 and C-Ideal-2 ................. 116
7-9 UNBAL: XTM vs. P-Ideal, C-Ideal-1 and C-Ideal-2 .............. 116
7-10 AQ: XTM vs. Diff-1, Diff-2, RR-1 And RR-2 ...................... 119
7-11 FIB: XTM vs. Diff-1, Diff-2, RR-1 And RR-2 ...................... 119
7-12 TSP: XTM vs. Diff-1, Diff-2, RR-1 And RR-2 ...................... 120
7-13 UNBAL: XTM vs. Diff-1, Diff-2, RR-1 And RR-2 ................... 120
7-14 Steal One vs. Steal Half ............................................. 123
7-15 AQ: Comparing Various Tree-Based Algorithms ............. 125
7-16 FIB: Comparing Various Tree-Based Algorithms .......... 125
7-17 TSP: Comparing Various Tree-Based Algorithms ............. 126
7-18 UNBAL: - Comparing Various Tree-Based Algorithms ...... 126
7-19 AQ(0.01): Variable $t_n$ ........................................... 130
7-20 FIB(20): Variable $t_n$ ........................................... 130
7-21 TSP(10): Variable $t_n$ ........................................... 131
7-22 UNBAL(16384): Variable $t_n$ ................................... 131
7-23 MATMUL (Coarse, Cached) ........................................ 135
7-24 MATMUL(64) (Coarse, Cached): Variable $t_n$ ............... 135
7-25 MATMUL (Fine, Cached) ........................................... 136
7-26 MATMUL(64) (Fine, Cached): Variable $t_n$ ............... 136
7-27 MATMUL (Coarse, Uncached) ..................................... 137
7-28 MATMUL(64) (Coarse, Uncached): Variable $t_n$ ........... 137
7-29 MATMUL (Fine, Uncached) ..................................... 138
7-30 MATMUL(64) (Fine, Uncached): Variable $t_n$ ........... 138
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 runtime. 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] Thread-Management 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 ac-
cessing 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.
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.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>Number of processors on a particular machine</td>
</tr>
<tr>
<td>$P_i$</td>
<td>Labeling for processor number $i$</td>
</tr>
</tbody>
</table>

Parameters for a $k$-ary $n$-cube Mesh Network

<table>
<thead>
<tr>
<th>$k$</th>
<th>Network Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>Network Dimensionality</td>
</tr>
<tr>
<td>$P_{i_0,i_1,...,i_{n-1}}$</td>
<td>Labeling for processor with mesh coordinates $i_0, i_1, ... i_{n-1}$; $0 \leq i_x \leq k - 1$</td>
</tr>
<tr>
<td>$t_n$</td>
<td>Network Speed: Cycles per Flit-Hop</td>
</tr>
</tbody>
</table>

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.


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 $\mu$ and variance $\sigma^2$, 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 $\sigma$. 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 \( \left\lfloor \frac{r}{p} \right\rfloor \) 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

\[ E = \frac{W}{pT} \]
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 algorithm's 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 \( N_l^I \) represents a node at level \( I \) in the tree residing on processor \( P_l \). Details of the mapping of tree nodes onto processors are given in Section 4.1.2.

Approximation 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.
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 an-
other 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 Thread Manager.

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 $O(nk)$, the expected cost is

\[ n \text{ is the mesh dimensionality; } k \text{ is the mesh radix.} \]
Figure 4-1: Binary X-Tree on a One-Dimensional Mesh Network: Labels of the form \( P_i \) indicate physical processors. Circles indicate nodes in the X-Tree. Thick lines indicate parent-child links. Thin lines indicate near-neighbor links.

actually \( \mathcal{O}\left(\frac{n}{d}\right) \), when the distributions of thread queue lengths are independent, identically distributed random variables with probabilities of non-zero length of at least \( \lambda \), for some \( \lambda : \operatorname{Prob}(\text{qlen} > 0) \geq \lambda \). 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 \( \mathcal{O}(nd) \) 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 \( \mathcal{O}(n) \)-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 $P_{i_0,i_1}$ 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:

$$N_{i_0,i_1,...}^l$$

where $l$ is the level of the node in the tree and $i_0, i_1, ...$ are the mesh coordinates of the processor on which the node resides ($P_{i_0,i_1,...}$). 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 \( N_{i_0,j_1,\ldots}^l \) and \( N_{j_0,j_1,\ldots}^m \) is written:

\[
D \left( N_{i_0,j_1,\ldots}^l, N_{j_0,j_1,\ldots}^m \right) = |j_0 - i_0| + |j_1 - i_1| + \ldots
\]

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:

\[
\mathcal{X} \left( N_{i_0,i_1,\ldots}^l, N_{j_0,j_1,\ldots}^m \right)
\]

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

The \( m \)th ancestor of node \( N_{i_0,i_1,\ldots}^0 \) is written:

\[
A_{i_0,i_1,\ldots}^m
\]

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

\[
A_{i_0,i_1,\ldots}^{l+m}
\]

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 \((N_{i_0, i_1, \ldots})\) and its parent \((A_{i_0, i_1, \ldots}^{i+1} = N_{i_0, i_1, \ldots}^{i+1})\) can be expressed through a set of (usually simple) embedding functions:

\[ i_0' = f_0 (i_0, i_1, \ldots, l); \quad i_1' = f_1 (i_0, i_1, \ldots, l); \quad \ldots \]

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 are used in formal algorithm statements and in mathematical proofs.
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:

\[ A_i^l = N_{i'}^l; \quad i' = i \land (-2^l) \]
Using this embedding, if a node on $P_i$ has an ancestor at level $l$, that ancestor is on $P_{i'}$, 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., $P_0$) 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:

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

Using this embedding, if a node on $P_i$ has an ancestor at level $l$, that ancestor is on $P_{i'}$, 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:

\[ D(N_i^j, N_j^m) = |j - i|. \]

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

\[ D(N_i^m, A_i^{m+1}) = 2^{m-1}, \]

except at the leaves of the tree.

The distance between a leaf node and its parent is:

\[ D(N_i^0, A_i^1) = \text{either } 0 \text{ or } 1, \]

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:

\[ D(N_i^m, N_i^{m\pm2^m}) = 2^m. \]
Two-Dimensional Case

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

\[ A_{i_0,i_1}^l = N_{i_0,j_0}^l; \quad i_0' = [i_0 \land (-2^{l-1})] \lor 2^{l-1}; \quad j_0' = [i_1 \land (-2^{l-1})] \lor 2^{l-1} \quad \text{(for } l \geq 1) \]

Using this embedding, if a node on \( P_{i,j} \) has an ancestor at level \( l \), that ancestor is on \( P_{i',j'} \). \( i' \) 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:

\[ D\left( N_{i_0,i_1}^l, N_{j_0,j_1}^m \right) = |j_0 - i_0| + |j_1 - i_1|. \]

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

\[ D\left( N_{i_0,i_1}^m, A_{i_0,i_1}^{m+1} \right) = 2 \left( 2^{m-1} \right), \]
except at the leaves of the tree.

The distance between a leaf node and its parent is:

\[ D \left( N_{i_0,i_1}^0, A_{i_0,i_1}^1 \right) = \text{either 0, 1 or 2}, \]

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 \( N_{0,1}^0 \) has five near neighbors: \( N_{0,0}^0, N_{1,0}^0, N_{1,1}^0, N_{1,2}^0, \) and \( N_{0,2}^0 \). \( N_{0,0}^0, N_{1,1}^0 \) and \( N_{0,2}^0 \) are edge-adjacent; \( N_{1,0}^0 \) and \( N_{1,2}^0 \) are vertex-adjacent.

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

\[ D \left( N_{i_0,i_1}^m, N_{i_0 \pm 2^m,i_1}^m \right) = D \left( N_{i_0,i_1}^m, N_{i_0,i_1 \pm 2^m}^m \right) = 2^m. \]

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

\[ D \left( N_{i_0,i_1}^m, N_{i_0 \pm 2^m,i_1 \pm 2^m}^m \right) = 2(2^m). \]

**n-dimensional Case**

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

\[
A_{i_0,i_1,...,i_n-1}^l = N_{i_0',i_1',...i_n-1}'
\]

\[
i_0' = [i_0 \land (-2^{l-1})] \lor 2^{l-1}
\]

\[
i_1' = [i_1 \land (-2^{l-1})] \lor 2^{l-1}
\]

\[
i_n-1' = [i_n-1 \land (-2^{l-1})] \lor 2^{l-1}
\]

Using this embedding, if a node on \( P_{i_0,i_1,...,i_n-1} \) has an ancestor at level \( l \), that ancestor is on \( P_{i_0',i_1',...i_n-1}' \). All the \( i_z' \)'s are obtained by taking the corresponding \( i_z \), 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:

$$D(N_{i_0,i_1,...,i_{n-1}}, N_{j_0,j_1,...,j_{n-1}}) = |j_0 - i_0| + |j_1 - i_1| + ... + |j_{n-1} - i_{n-1}|.$$ 

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

$$D(N^m_{i_0,i_1,...,i_{n-1}}, A^{m+1}_{i_0,i_1,...,i_{n-1}}) = n(2^{m-1}),$$

except at the leaves of the tree.

The distance between a leaf node and its parent is:

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

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:

$$\binom{n}{i} 2^i.$$ 

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:

$$i(2^m).$$

### 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.
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 $N_i$. 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 $P_i$ 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.
We start with processors $P_0$, $P_1$, $P_2$, $P_3$, $P_4$, $P_6$ and $P_7$ busy and processor $P_5$ idle. All runnable thread queues are empty except for those on $P_0$ and $P_1$, which contain six and three threads, respectively.
P₅, which is idle, initiates a thread search. The search process first examines P₅'s leaf node (N₅⁰), where it finds no work. It then examines N₅⁰'s two nearest neighbors, N₄⁰ and N₆⁰, finding both of their presence bits to be zero, so it continues one level higher in the tree with node N₅¹. N₅¹'s presence bit is zero, as are those of both its neighbors, N₃¹ and N₇¹, so the search continues one level higher at node N₆². N₆²'s presence bit is zero, but its neighbor, N₇² has its presence bit set, so the search goes into gathering mode.
The search process that originated on P5 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 $N^0_1$ and $N^0_0$, both children of $N^1_1$ and both of which have set presence bits.

Meanwhile, P6 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: $N^0_6$, $N^0_5$, $N^0_7$, $N^1_1$ and $N^1_5$, finding them all to be zero. When it gets to node $N^2_6$, 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 $S_6$. 
The gathering process initiated at node $N_2$ 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 $P_0$’s and $P_1$’s queues to be detached and sent back up the tree towards the requester.

Meanwhile, yet another processor ($P_7$) has become idle. It initiates a third thread search process, which examines the presence bits of nodes $N_0^0$ and $N_0^1$, which are both still zero. When it gets to node $N_1^1$, it waits for the second searcher to return with work. The waiting searcher is indicated in the figure by the symbol $S_7$. 
The gather process continues. The threads taken from processors $P_0$ and $P_1$ are combined at node $N_1^1$ and sent on up to node $N_2^2$, where the gathering process started.
The threads are sent back to node $N_2^2$, 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.
Searcher $S_5$ sends its threads one link closer to $P_5$. 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.
A tree update process is sent out from $P_5$ 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 $P_i$ and $P_j$ is simply $|j - i|:

$$d = D(N^0_i, N^0_j) = |j - i|$$

This is the lowest possible cost for communicating between $P_i$ and $P_j$.

Also, when following the shortest path through the tree between nodes $N^0_i$ and $N^0_j$, 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 $N^0_i$ 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:

$$
X(N_i^0, N_j^0) = D(N_i^0, A_i^l) + D(A_i^l, A_j^l) + D(A_j^l, N_j^0)
$$

$$
\leq \left( 1 + \sum_{x=1}^{l-1} 2^{x-1} \right) + 2^l + \left( 1 + \sum_{x=1}^{l-1} 2^{x-1} \right)
$$

$$
\leq 2^{l-1} + 2^l + 2^{l-1}
$$

$$
\leq 2 \left( 2^l \right)
$$

$$
\leq 2 \left( 2^{\log_2 d} \right)
$$

$$
\leq 4d
$$

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 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 (N7) to node B (N12) covers five network hops. The corresponding tree traversal covers 15 network hops: N7 → N1 → N6 → N12 → N14 → N12.

The destination processor then sends some of its work back to the requester. If the requesting processor is P1 and the destination processor is Pj, 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.

\[
B(N_i^0, N_j^0) \leq \left( 1 + \sum_{z=1}^{l-1} 2^{z-1} \right) + 2^l + \left( 1 + \sum_{z=0}^{l-1} 2^{l-z} 2^{z-1} \right) + 2^l + \left( 1 + \sum_{x=1}^{l-1} 2^{x-1} \right)
\]
\[
\leq 2^{l-1} + 2^l + l2^{l-1} + 2^l + 2^{l-1}
\]
\[
\leq (l + 6) \left( 2^{l-1} \right)
\]
\[
\leq (7 + \log_2 d) \left( 2^{[\log_2 d] - 1} \right)
\]
\[
\leq (7 + \log_2 d) d
\]

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

65
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 $P_i$ and a different node in the X-Tree.

$$
\mathcal{K}(N_i^0, N_j^0) = \sum_{x=1}^{l} \mathcal{D}(N_i^0, A_i^x) + \sum_{x=l}^{0} \mathcal{D}(N_i^0, A_j^x)
$$

$$
\geq -1 + \left( \sum_{x=0}^{l-1} 2^x \right) + \left( \sum_{x=l}^{0} 2^{l-1} \right)
$$

$$
\geq (l + 1)2^{l-1} - 1
$$

$$
\geq (\lceil \log_2 d \rceil + 1)2^{\lceil \log_2 d \rceil - 1} - 1
$$

$$
\geq (\log_2 d) \frac{d}{4} - 1
$$

In other words, for the shared-memory case, the cost is $\Omega(d \log d)$. 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: $U$, the worst-case tree update critical-path cost, can be as bad as $O(k)$. We now show why this is the case.

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

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: \( \lambda \). 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 \( \lambda \), for some \( \lambda : \text{Prob} (qlen > 0) \geq \lambda \). In the following derivation, \( E[U] \) 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 \( C(i) \) is the expected cost of communicating between a node at the \( i \)th level and its parent. Finally, let \( \mu = (1 - \lambda) \).

\[
E[U] = \sum_{i=0}^{L-2} C(i) \text{Prob} (\text{lev} = i)
\]

\[
\leq 2^{-1} \left[ \mu^{(2^0)} - \mu^{(2^1)} \right] + 2^0 \left[ \mu^{(2^1)} - \mu^{(2^2)} \right] + \ldots + 2^{L-4} \left[ \mu^{(2^{L-3})} - \mu^{(2^{L-2})} \right] + 2^{L-3} [\mu^{(2^{L-2})}]
\]

\[
\leq \mu 2^{-2} + \sum_{i=0}^{L-2} 2^{i-2} \mu^{(2^i)}
\]

\[
\leq \frac{1}{4} \left[ \mu + \sum_{i=0}^{L-2} 2^i \mu^{(2^i)} \right]
\]

\[
\leq \frac{1}{4} \left[ \mu + \sum_{j=1}^{\infty} j \mu^j \right]
\]

\[
\leq \frac{\mu}{4} \left[ 1 + \frac{d}{d\mu} \sum_{j=0}^{\infty} \mu^j \right]
\]

\[
\leq \frac{\mu}{4} \left[ 1 + \frac{1}{\mu} \frac{1}{1 - \mu} \right]
\]

\[
\leq \frac{\mu}{4} \left[ 1 + \frac{1}{(1 - \mu)^2} \right]
\]

\[
\leq \frac{(1 - \lambda)}{4} \left[ 1 + \frac{1}{\lambda^2} \right]
\]

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 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.

$$B(\mathcal{U}) \leq k + \sum_{i=0}^{L-1} 2 \left(2^i\right)$$

$$\leq k + 2 \left(2^{\lceil \log_2 k \rceil}\right)$$

$$\leq k + 4k$$

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:

$$E[B(\mathcal{U})] = E[\mathcal{U}] + E[\mathcal{B}(\text{neighbor-cache update})]$$

$$\leq \frac{(1 - \lambda)}{4} \left[1 + \frac{1}{\lambda^2}\right] + \sum_{i=0}^{L-1} 2 \left(2^i\right) \mu^{(\infty)}$$

$$\leq \frac{(1 - \lambda)}{4} \left[1 + \frac{1}{\lambda^2}\right] + 2\mu \frac{d}{d\mu} \frac{1}{1 - \mu}$$

$$\leq \frac{(1 - \lambda)}{4} \left[1 + \frac{1}{\lambda^2}\right] + \frac{2(1 - \lambda)}{\lambda^2}$$

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 $n$. 

70
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.

\[
U \leq 1 + \sum_{i=1}^{L-1} 2^i \\
\leq 2^L \\
\leq 2^\lceil \log_2 k \rceil \\
\leq 2k
\]

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 \( P_{i_0, i_1} \) and \( P_{j_0, j_1} \) is simply \(|j_0 - i_0| + |j_1 - i_1|\):

\[
d = D(N^0_{i_0, i_1}, N^0_{j_0, j_1}) = |j_0 - i_0| + |j_1 - i_1|
\]

This is the lowest possible cost for communicating between \( P_{i_0, i_1} \) and \( P_{j_0, j_1} \).

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:

\[
\max (\log_2 |j_0 - i_0|, \log_2 |j_1 - i_1|) \leq l \leq \max (\log_2 |j_0 - i_0|, \log_2 |j_1 - i_1|)
\]

\[
\max (\log_2 |j_0 - i_0|, \log_2 |j_1 - i_1|) \leq [\log_2 d] \leq [\log_2 \log_2 d]
\]

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:

\[
[\log_2 k] \leq L \leq [\log_2 k]
\]
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:

$$
\mathcal{X}(N^0_{i_0,i_1}, N^0_{j_0,j_1}) = D(N^0_{i_0,i_1}, A^l_{i_0,i_1}) + D(A^l_{i_0,i_1}, A^l_{j_0,j_1}) + D(A^l_{j_0,j_1}, N^0_{j_0,j_1})
$$

$$
\leq \left[ 1 + \sum_{x=1}^{l-1} 2 \left( 2^{x-1} \right) \right] + 2 \left( 2^{l} \right) + \left[ 1 + \sum_{x=1}^{l-1} 2 \left( 2^{x-1} \right) \right]
$$

$$
\leq 2 \left( 2^{l-1} + 2^l + 2^{l-1} \right)
$$

$$
\leq 4 \left( 2^l \right)
$$

$$
\leq 4 \left( 2^{\log_2 d} \right)
$$

$$
\leq 8d
$$

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 \(N_{16,15}^0\) to node B \(N_{15,24}^0\) covers ten network hops. The corresponding tree traversal covers 62 network hops: \(N_{16,15}^0 \rightarrow N_{17,15}^1 \rightarrow N_{18,14}^2 \rightarrow N_{20,12}^3 \rightarrow N_{24,8}^4 \rightarrow N_{28,15}^5 \rightarrow N_{28,28}^6 \rightarrow N_{28,28}^7 \rightarrow N_{15,25}^8 \rightarrow N_{15,24}^9\).

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 $N_{i_0,i_1}^0$ when the closest work is at $N_{j_0,j_1}^0$:

$$B(N_{i_0,i_1}^0, N_{j_0,j_1}^0) \leq \left(1 + \sum_{x=1}^{l-1} 2 (2^{x-1})\right) + 2 \left(2^l\right) + \left[1 + \sum_{x=0}^{l-1} (4^{l-x}) (2) (2^{x-1})\right]$$

$$\leq 2 \left(2^{l-1} + 2^l + 4^l - 1\right)$$

$$\leq 2 (l + 3) (4^{l-1})$$

$$\leq 2 \left[3 + \lfloor \log_2 d \rfloor\right] \left[4^\lfloor \log_2 d \rfloor - 1\right]$$

$$\leq 2 \left[4 + \log_2 d\right] d^2$$

Although the latency for the search is $O(d)$, where $d$ is the distance between the searcher and the nearest work, the network bandwidth consumed is $O(d^2 \log d)$. This is not surprising, due to the use of a gathering process to collect threads from the entire sub-tree rooted at $A_{j_0,j_1}^1$.

**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 $O(k)$. We now show why this is the case:

$$U \leq 2 \left( 1 + \sum_{i=0}^{L-2} 2^i \right)$$

$$\leq 2 \left( 2^{L-1} \right)$$

$$\leq 2 \left( 2^{\log_2 k - 1} \right)$$

$$\leq 2k$$

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 \( \lambda \), for some \( \lambda : \text{Prob}(qlen > 0) \geq \lambda \).

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

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 \( O\left( \frac{1}{\lambda^2} \right) \), 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.

\[ \mathcal{B}(U) \leq k + \sum_{i=0}^{L-1} (3^2 - 1) 2^i \]
\[ \leq k + 8 \left(2^{\lfloor \log_2 k \rfloor} \right) \]
\[ \leq k + 16k \]

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

The expected bandwidth requirement is significantly lower:

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

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 \textit{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 \(P_{i_0,i_1,\ldots,i_{n-1}}\) and \(P_{j_0,j_1,\ldots,j_{n-1}}\) is simply \(|j_0 - i_0| + |j_1 - i_1| + \ldots + |j_{n-1} - i_{n-1}|\):

\[
d = D\left(N^0_{i_0,i_1,\ldots,i_{n-1}}, N^0_{j_0,j_1,\ldots,j_{n-1}}\right) = |j_0 - i_0| + |j_1 - i_1| + \ldots + |j_{n-1} - i_{n-1}|
\]

This is the lowest possible cost for communicating between \(P_{i_0,i_1,\ldots,i_{n-1}}\) and \(P_{j_0,j_1,\ldots,j_{n-1}}\).

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

\[
\max (\lfloor \log_2 |j_0 - i_0| \rfloor, \ldots, \lfloor \log_2 |j_{n-1} - i_{n-1}| \rfloor) \leq l \leq \max (\lfloor \log_2 |j_0 - i_0| \rfloor, \ldots, \lfloor \log_2 |j_{n-1} - i_{n-1}| \rfloor)
\]

\[
\lfloor \max (\log_2 |j_0 - i_0|, \ldots, \log_2 |j_{n-1} - i_{n-1}|) \rfloor \leq l \leq \lfloor \max (\log_2 |j_0 - i_0|, \ldots, \log_2 |j_{n-1} - i_{n-1}|) \rfloor
\]

\[
\lfloor \log_2 d \rfloor \leq l \leq \lfloor \log_2 d \rfloor
\]

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:

\[
\lfloor \log_2 k \rfloor \leq L \leq \lfloor \log_2 k \rfloor
\]
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 \( N^0_{i_0,i_1,...,i_{n-1}} \) to node \( N^0_{j_0,j_1,...,j_{n-1}} \) is established by ascending \( l \) levels in the tree from node \( N^0_{i_0,i_1,...,i_{n-1}} \) to node \( A^l_{i_0,i_1,...,i_{n-1}} \), crossing over to node \( A^l_{j_0,j_1,...,j_{n-1}} \) and descending back down to node \( N^0_{j_0,j_1,...,j_{n-1}} \). The following analysis demonstrates the competitive factor of \( n \):

\[
X(N^0_{i_0,i_1,...,i_{n-1}}, N^0_{j_0,j_1,...,j_{n-1}}) = D(N^0_{i_0,i_1,...,i_{n-1}, A^l_{i_0,i_1,...,i_{n-1}}}) \\
+ D(A^l_{i_0,i_1,...,i_{n-1}, A^l_{j_0,j_1,...,j_{n-1}}}) \\
+ D(A^l_{j_0,j_1,...,j_{n-1}, N^0_{j_0,j_1,...,j_{n-1}}}) \\
\leq \left[ 1 + \sum_{z=1}^{l-1} n (2^{z-1}) \right] + n (2^l) + \left[ 1 + \sum_{z=1}^{l-1} n (2^{z-1}) \right] \\
\leq n (2^{l-1} + 2^l + 2^{l-1}) \\
\leq 4n (2^{l-1}) \\
\leq 4n (2^{\log_2 d[l]} - 1) \\
\leq 4nd
\]

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 \( N_{i_0,i_1,...,i_{n-1}} \) when the closest work is at \( N_{j_0,j_1,...,j_{n-1}} \):

\[
B \left( N_{i_0,i_1,...,i_{n-1}}, N_{j_0,j_1,...,j_{n-1}} \right) \leq \left( 1 + \sum_{x=1}^{i-1} n \left( 2^x - 1 \right) \right) + n \left( 2^l \right) + \left[ 1 + \sum_{x=0}^{l-1} (2^n)^{l-x} (n) \left( 2^{x-1} \right) \right] \\
\leq n \left[ 2^{l-1} + 2^l + l (2^n)^{l-1} \right] \\
\leq n \left[ l + 3 \left( (2^n)^{l-1} \right) \right] \\
\leq n \left[ 3 + \lfloor \log_2 d \rfloor \left( (2^n)^{\lfloor \log_2 d \rfloor - 1} \right) \right] \\
\leq n \left[ 4 + \log_2 d \right] d^n
\]

Although the running time for the search is \( O(n d) \), where \( d \) is the distance between the searcher and the nearest work, the network bandwidth consumed is \( O(n d^m \log c) \). Again, this is expected behavior, due to the use of a gathering process to collect threads from the entire sub-tree rooted at \( A_{j_0,j_1,...,j_{n-1}} \).
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 $O(nk)$. We now show why this is the case.

\[
U \leq n \left( 1 + \sum_{i=0}^{L-2} 2^i \right) \\
\leq n 2^{L-1} \\
\leq n 2^{[\log_2 k]-1} \\
\leq nk
\]

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 \( \lambda \), for some \( \lambda : \text{Prob} (\text{qlen} > 0) \geq \lambda \).

\[
\mathbb{E}[U] = \sum_{i=0}^{L-2} C(i_0, i_1, \ldots, i_{n-1}) P(i_0, i_1, \ldots, i_{n-1})
\]

\[
\leq n \left[ 2^{L-1} (\mu[(2^n)^0] - \mu[(2^n)^1]) + 2^0 (\mu[(2^n)^1] - \mu[(2^n)^2])
\right.

\[
+ \ldots + 2^{L-4} (\mu[(2^n)^L] - \mu[(2^n)^{L-2}]) + 2^{L-3} (\mu[(2^n)^{L-2}])
\right]
\]

\[
\leq n \left( \mu 2^{-2} + \sum_{i=0}^{L-2} 2^{i-2} \mu[(2^n)^i] \right)
\]

\[
\leq \frac{n}{4} \left( \mu + \sum_{i=0}^{L-2} 2^i \mu[(2^n)^i] \right)
\]

\[
\leq \frac{n}{4} \left( \mu + \sum_{i=0}^{\infty} (2^n)^i \mu[(2^n)^i] \right)
\]

\[
\leq \frac{n}{4} \left[ \mu + \sum_{j=1}^{\infty} j \mu^j \right]
\]

\[
\leq \frac{\mu n}{4} \left[ 1 + \frac{1}{(1 - \mu)^2} \right]
\]

\[
\leq \frac{n (1 - \lambda)}{4} \left[ 1 + \frac{1}{\lambda^2} \right]
\]

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 \( O \left( \frac{n}{\lambda^2} \right) \), 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:

\[ B(U) \leq k + \sum_{i=0}^{L-1} (3^n - 1) 2^i \]
\[ \leq k + (3^n - 1) \left(2^{\lceil \log_2 k \rceil}\right) \]
\[ \leq k + (3^n - 1) k \]

The worst-case bandwidth requirement for the tree update process is \( O(3^n k) \).

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

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

The expected bandwidth requirement is \( O\left(\frac{3^n}{\lambda^2}\right) \).
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-
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
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.
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 \( \text{proc} \) when it first wakes up. When \( \text{proc} \) 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-
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.

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

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

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

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 \( \gamma \) 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 \( r = \frac{(o-1)n+5}{6} \), not \( r = \frac{(o-1)n+3}{6} \) 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.
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 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 1-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.
<table>
<thead>
<tr>
<th>Application</th>
<th>$T_1$</th>
<th>$T_{crit}$</th>
<th>No. Threads</th>
<th>Avg. Grain Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIB(15)</td>
<td>652,439</td>
<td>4,495</td>
<td>1,219</td>
<td>535</td>
</tr>
<tr>
<td>FIB(20)</td>
<td>7,244,462</td>
<td>6,110</td>
<td>13,529</td>
<td>535</td>
</tr>
<tr>
<td>FIB(25)</td>
<td>80,350,904</td>
<td>7,725</td>
<td>150,889</td>
<td>533</td>
</tr>
<tr>
<td>AQ(0.5)</td>
<td>437,191</td>
<td>11,725</td>
<td>309</td>
<td>1,415</td>
</tr>
<tr>
<td>AQ(0.1)</td>
<td>2,141,754</td>
<td>13,480</td>
<td>1,513</td>
<td>1,416</td>
</tr>
<tr>
<td>AQ(0.05)</td>
<td>4,169,108</td>
<td>13,480</td>
<td>2,943</td>
<td>1,417</td>
</tr>
<tr>
<td>AQ(0.01)</td>
<td>20,201,061</td>
<td>16,990</td>
<td>14,269</td>
<td>1,416</td>
</tr>
<tr>
<td>AQ(0.005)</td>
<td>43,062,592</td>
<td>16,990</td>
<td>30,417</td>
<td>1,416</td>
</tr>
<tr>
<td>AQ(0.001)</td>
<td>213,632,152</td>
<td>20,020</td>
<td>150,049</td>
<td>1,424</td>
</tr>
<tr>
<td>TSP(8)</td>
<td>3,635,870</td>
<td>4,760</td>
<td>$\approx$ 4,000</td>
<td>900</td>
</tr>
<tr>
<td>TSP(9)</td>
<td>12,613,984</td>
<td>5,605</td>
<td>$\approx$ 15,000</td>
<td>850</td>
</tr>
<tr>
<td>TSP(10)</td>
<td>41,708,532</td>
<td>6,500</td>
<td>$\approx$ 50,000</td>
<td>830</td>
</tr>
<tr>
<td>TSP(11)</td>
<td>326,034,362</td>
<td>7,445</td>
<td>$\approx$ 400,000</td>
<td>810</td>
</tr>
<tr>
<td>UNBAL(1024)</td>
<td>601,451</td>
<td>587</td>
<td>1,024</td>
<td>587</td>
</tr>
<tr>
<td>UNBAL(4096)</td>
<td>2,404,715</td>
<td>587</td>
<td>4,096</td>
<td>587</td>
</tr>
<tr>
<td>UNBAL(16384)</td>
<td>9,617,771</td>
<td>587</td>
<td>16,384</td>
<td>587</td>
</tr>
<tr>
<td>UNBAL(65536)</td>
<td>38,469,995</td>
<td>587</td>
<td>65,636</td>
<td>587</td>
</tr>
</tbody>
</table>

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.
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.
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.
<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>( t_n )</th>
<th>( p ) Range</th>
<th>Managers</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQ</td>
<td>0.01</td>
<td>2</td>
<td>1-16384</td>
<td>all but Stat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1-16384</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1-16384</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td>FIB</td>
<td>20</td>
<td>2</td>
<td>1-16384</td>
<td>all but Stat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1-16384</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1-16384</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td>TSP</td>
<td>10</td>
<td>2</td>
<td>1-4096</td>
<td>all but Stat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td>UNBAL</td>
<td>16384</td>
<td>2</td>
<td>1-16384</td>
<td>all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1-16384</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1-16384</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>1-4096</td>
<td></td>
</tr>
<tr>
<td>MATMUL (coarse, cached)</td>
<td>64</td>
<td>2</td>
<td>1-1024</td>
<td>all but Ideal and XTM-C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1-1024</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1-1024</td>
<td></td>
</tr>
<tr>
<td>MATMUL (coarse, uncached)</td>
<td>64</td>
<td>2</td>
<td>1-1024</td>
<td>all but Ideal and XTM-C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1-1024</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1-1024</td>
<td></td>
</tr>
<tr>
<td>MATMUL (fine, cached)</td>
<td>64</td>
<td>2</td>
<td>1-1024</td>
<td>all but Ideal and XTM-C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1-1024</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1-1024</td>
<td></td>
</tr>
<tr>
<td>MATMUL (fine, uncached)</td>
<td>64</td>
<td>2</td>
<td>1-1024</td>
<td>all but Ideal and XTM-C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1-1024</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1-1024</td>
<td></td>
</tr>
</tbody>
</table>

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.
### Table 7.3: Regions of Interest

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

$T_1$ 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.
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).
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-I 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 \(\Theta(l^2)\), 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
the 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).
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.
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 thinnest 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 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: \(\text{AQ}(0.01)\): Variable \(t_n\).

Figure 7-20: \(\text{FIB}(20)\): Variable \(t_n\).
Figure 7-21: TSP(10): Variable $t_n$.

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

131
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 $\Delta X$ and $\Delta t$ 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 disappear, 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 Statalways 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 $t_n$. 

135
Figure 7-25: \textit{MATMUL (Fine,Cached).}

Figure 7-26: \textit{MATMUL(64) (Fine, Cached): Variable }t_n.
Figure 7-27: MATMUL (Coarse, Uncached). Figure 7-28: MATMUL(64) (Coarse, Uncached): Variable $t_n$. 

137
Figure 7-29: MATMUL (Fine, Uncached).

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

138
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 dis-
tributed, 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 polylog-
competitive 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* ((RO (* X Y))
         (R1 (* RO RO)))
    (* R1 R1)))
(DEFINE (AQ XO YO X1 Y1 TOLERANCE Q0)
  (LET* ((XM (/ (+ XO X1) 2.0)))
         (YM (/ (+ YO Y1) 2.0))
         (Q1 (Q XO YO XM YM))
         (Q2 (Q XM YO X1 YM))
         (Q3 (Q XO 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 XO YO XM YM TOLERANCE Q1))))
              (SUM1 (BLOCK (PAUSE 80)
                           (FUTURE (AQ XM YO X1 YM TOLERANCE Q2))))
              (SUM2 (BLOCK (PAUSE 80)
                           (FUTURE (AQ XO YM XM Y1 TOLERANCE Q3))))
              (SUM3 (BLOCK (PAUSE 80)
                           (FUTURE (AQ XM YM X1 Y1 TOLERANCE Q4))))
              (VAL (BLOCK (PAUSE 100)
                           (+ (TOUCH SUMO) (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))
(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)))

147
;;; 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 . 4) (-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))))

152
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)))
(define unordered-ic-d-mats
  '(
    #(0)
    #(0 1)
      #(0 5 3)
        #(0 2 4 5)
          #(0 3 7 7 6)
            #(0 5 8 7 11 16)
              #(5 0 2 2 10 3 5)

    #((0 5 0 4)
      #((0 3 4 0))
      #*(0 2 4 5)
        #((0 2 3)
          #((0 4 2 0 2)
            #((0 5 3 2 0))
            #((0 3 7 7 6)
              #((0 3 0 5 5 3)
                #((0 7 5 0 2 3)
                  #((0 7 5 2 0 2)
                    #((0 6 3 3 2 0))
                    #((0 5 5 8 7 11 16)
                      #((0 5 0 6 4 10 13)
                        #((0 8 6 0 2 4 8)
                          #((0 7 4 2 0 6 10)
                            #((0 11 10 4 6 0 7)
                              #((0 16 13 8 10 7 0))
                              #((0 5 2 2 10 3 5)
                                #((0 2 7 0 5 10 4 2)
                                  #((0 2 3 5 0 9 5 7)
                                    #((0 10 11 0 9 0 13 11)
                                      #((0 3 6 4 5 13 0 6)
                                        #((0 5 10 2 7 11 6 0))
                                        #((0 10 7 4 8 3 5 8)
                                          #((0 10 0 5 11 13 11 14 5)
                                            #((0 7 5 0 9 8 8 9 2)
                                              #((0 4 11 9 0 13 1 9 11)
                                                #((0 8 13 8 13 0 11 5 8)
                                                  #((0 3 11 8 1 11 0 7 10)
                                                    #((0 5 14 9 5 7 0 9)
                                                      #((0 8 5 2 11 8 10 9 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 6)
        #(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 11 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 6 2 0 5)
        #(10 11 11 14 13 5 5 0))
  )

158
8(*(()051383391314)8(*(()5012877141819)8(*(()1312051111141517)8(*(()885066101314)8(*(()371160071112)8(*(()371160071112)8(*(()914141077045)8(*(()131815131114402)8(*(()141917141212520)8(*(()05157812171519)8(*(()50510121116221820)8(*(()15057913181620)8(*(()7127201011161623)8(*(()81199100510713)8(*(()1216131211505514)8(*(()17221817161050716)8(*(()15181616757709)8(*(()192020232313141690))8(*(()045697713131620)8(*(()407913101017182024)8(*(()57029101115151920)8(*(()69209111215151919)8(*(()91399079661011)8(*(()710101702781016)8(*(()71011129208101117)8(*(()1317151567780249)8(*(()1318151568102047)8(*(()162019191010114409)8(*(()202420191116179790))8(*(()01518202022181419161421)8(*(()15035574611151624)8(*(()183022451013192028)8(*(()205202361215212230)8(*(()20520202441013192129)8(*(()227432051113202229)8(*(()1845645068151724)8(*(()1461012101160591018)8(*(()19111315131385081117)8(*(()1615192119201598039)8(*(()141620222122171011308)8(*(()212428302929241817980))))))
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-i (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-i) 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))))))))

160
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))))))))

161
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))))

162
;;; the following code is snarfed from dmatrix.t
;;;
(define-integrable (%my-ceiling x y)
  (fx/ (fx+ x (fx- y i)) 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)))
        (do ((i width) ((fx= i width))
           (let ((vec (make-vector 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)))))
(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)))
                (*1-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)))))
(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 1-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 *invv-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}"
(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 (fxashr *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 1)
  (pause-write-from-read-time x 1 %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))
          ((fx>= pid *N-Processors*)
            (pause (+ (transit-time *inwr-msg-size* hpid max-pid)
                      *process-inwr-cycles*
                      (transit-time *inwr-ack-msg-size* max-pid hpid)
                      *process-inwr-ack-cycles*
                      (* (fx- n 1) *inwr-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 ((ml (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 ((ml (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DLMATRIX i k 0)))
    (message "finished initialization!"
     (matmul ml m2 m3)
     m3))

(define (matmul ml 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 ml 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*))))

177
Fine-Grained, Static, With Caches

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

(define (matmul m1 m2 m3)
  (let ((n *Procs-Per-Dime*))
    (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))

179
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-vidth (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*))))))))
(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)))
    (do ((i x-height) ((x-width 0)))
      (do ((j y-width) ((y-width 0)))
        (set (LMATREF z i j) (fx+ (LMATREF z i j)
                                 (acc (k 0 x-width)
                                      (fx+ (MATREF x i k) (MATREF y k j))))))
    (pause (+ *mul-cycles* *add-cycles* *lmatref-cycles*))))
Fine-Grained, Static, No Caches

(define (main i j k)
  (let ((ml (MAKE-DMATRIX i j 1))
        (m2 (MAKE-DMATRIX j k 2))
        (m3 (MAKE-DLMATRIX i k 0)))
    (message "finished initialization!")
    (matmul m1 m2 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)))
    (do-times (i x-height)
      (pause *loop-cycles*)
      (do-times (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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>435035</td>
<td>112062</td>
<td>34129</td>
<td>15087</td>
<td>14005</td>
<td>14005</td>
<td>14005</td>
<td>14005</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>437191</td>
<td>118076</td>
<td>36085</td>
<td>18346</td>
<td>16616</td>
<td>16627</td>
<td>16627</td>
<td>16627</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>437209</td>
<td>115735</td>
<td>35832</td>
<td>18254</td>
<td>16467</td>
<td>16970</td>
<td>17304</td>
<td>17432</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>437209</td>
<td>144119</td>
<td>36058</td>
<td>16989</td>
<td>16942</td>
<td>16875</td>
<td>17279</td>
<td>17416</td>
</tr>
<tr>
<td>RR-1</td>
<td>437191</td>
<td>117456</td>
<td>40417</td>
<td>19950</td>
<td>17684</td>
<td>18907</td>
<td>18694</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>437191</td>
<td>116304</td>
<td>37367</td>
<td>21135</td>
<td>17932</td>
<td>18534</td>
<td>18539</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>445119</td>
<td>126182</td>
<td>49405</td>
<td>44363</td>
<td>45505</td>
<td>45523</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>445119</td>
<td>126875</td>
<td>46455</td>
<td>33036</td>
<td>34017</td>
<td>34019</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>437191</td>
<td>119230</td>
<td>39386</td>
<td>21287</td>
<td>18493</td>
<td>18410</td>
<td>18385</td>
<td>20405</td>
</tr>
<tr>
<td>XTM</td>
<td>437191</td>
<td>116045</td>
<td>40749</td>
<td>22285</td>
<td>25064</td>
<td>24280</td>
<td>21566</td>
<td>21999</td>
</tr>
<tr>
<td>XTM-C</td>
<td>437191</td>
<td>121966</td>
<td>47901</td>
<td>26724</td>
<td>31433</td>
<td>32781</td>
<td>29542</td>
<td>29138</td>
</tr>
</tbody>
</table>

Table C.1: AQ(0.5) – Running Times (cycles).
<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>2131170</td>
<td>542674</td>
<td>141886</td>
<td>43030</td>
<td>19226</td>
<td>16149</td>
<td>16149</td>
<td>16149</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>2141754</td>
<td>564540</td>
<td>150438</td>
<td>45283</td>
<td>22013</td>
<td>19394</td>
<td>19443</td>
<td>19443</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>2141772</td>
<td>542714</td>
<td>145202</td>
<td>47139</td>
<td>23980</td>
<td>19464</td>
<td>20160</td>
<td>20465</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>2141772</td>
<td>542502</td>
<td>145551</td>
<td>46704</td>
<td>23980</td>
<td>19464</td>
<td>20188</td>
<td>20801</td>
</tr>
<tr>
<td>RR-1</td>
<td>2141754</td>
<td>545377</td>
<td>150423</td>
<td>53676</td>
<td>28722</td>
<td>28662</td>
<td>27506</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>2141754</td>
<td>544633</td>
<td>149842</td>
<td>52986</td>
<td>23980</td>
<td>18991</td>
<td>18991</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>2180964</td>
<td>609435</td>
<td>173445</td>
<td>68067</td>
<td>63054</td>
<td>63054</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>2180964</td>
<td>609435</td>
<td>173445</td>
<td>68067</td>
<td>63054</td>
<td>63054</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>2141754</td>
<td>551045</td>
<td>156336</td>
<td>53321</td>
<td>28386</td>
<td>24167</td>
<td>25218</td>
<td>25497</td>
</tr>
<tr>
<td>XTM</td>
<td>2141754</td>
<td>550285</td>
<td>152832</td>
<td>54432</td>
<td>39297</td>
<td>29941</td>
<td>27228</td>
<td>31589</td>
</tr>
<tr>
<td>XTM-C</td>
<td>2147417</td>
<td>560525</td>
<td>158993</td>
<td>60323</td>
<td>28662</td>
<td>25213</td>
<td>25497</td>
<td>56160</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>418500</td>
<td>1054102</td>
<td>270814</td>
<td>73886</td>
<td>27454</td>
<td>16150</td>
<td>16150</td>
<td>16149</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>4180434</td>
<td>1059581</td>
<td>283181</td>
<td>77950</td>
<td>30024</td>
<td>19828</td>
<td>18991</td>
<td>19330</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>4169126</td>
<td>1050843</td>
<td>279758</td>
<td>80901</td>
<td>40828</td>
<td>35272</td>
<td>33973</td>
<td>33973</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>4169126</td>
<td>1049507</td>
<td>276636</td>
<td>87606</td>
<td>38258</td>
<td>40102</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>RR-1</td>
<td>4169108</td>
<td>1055143</td>
<td>279758</td>
<td>89091</td>
<td>40828</td>
<td>35272</td>
<td>33973</td>
<td>33973</td>
</tr>
<tr>
<td>RR-2</td>
<td>4169108</td>
<td>1054012</td>
<td>276636</td>
<td>87606</td>
<td>38258</td>
<td>40102</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>4244440</td>
<td>1147785</td>
<td>310130</td>
<td>119954</td>
<td>125941</td>
<td>122621</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>4244440</td>
<td>1178724</td>
<td>320553</td>
<td>102799</td>
<td>101305</td>
<td>86826</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>4180434</td>
<td>1066320</td>
<td>284062</td>
<td>91730</td>
<td>38763</td>
<td>29169</td>
<td>25684</td>
<td>25957</td>
</tr>
<tr>
<td>XTM</td>
<td>4169108</td>
<td>1055502</td>
<td>287066</td>
<td>87052</td>
<td>50737</td>
<td>33773</td>
<td>34173</td>
<td>33661</td>
</tr>
<tr>
<td>XTM-C</td>
<td>4169108</td>
<td>1076687</td>
<td>303661</td>
<td>97568</td>
<td>61870</td>
<td>57997</td>
<td>68921</td>
<td>66078</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>20101185</td>
<td>5092266</td>
<td>1292960</td>
<td>338377</td>
<td>92863</td>
<td>31235</td>
<td>20438</td>
<td>20438</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>20212387</td>
<td>5298344</td>
<td>1346799</td>
<td>344455</td>
<td>96475</td>
<td>36430</td>
<td>25419</td>
<td>25218</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>20201079</td>
<td>5069478</td>
<td>1282056</td>
<td>341638</td>
<td>98190</td>
<td>36878</td>
<td>26236</td>
<td>26977</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>20201079</td>
<td>5063978</td>
<td>1260377</td>
<td>333620</td>
<td>97802</td>
<td>34671</td>
<td>25788</td>
<td>26977</td>
</tr>
<tr>
<td>RR-1</td>
<td>20201061</td>
<td>5062577</td>
<td>1287129</td>
<td>356119</td>
<td>116679</td>
<td>81243</td>
<td>81555</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>20201061</td>
<td>5062895</td>
<td>1268787</td>
<td>351224</td>
<td>119718</td>
<td>77522</td>
<td>73782</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>20562663</td>
<td>5531705</td>
<td>1448165</td>
<td>410432</td>
<td>297889</td>
<td>316372</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>20562663</td>
<td>5690009</td>
<td>1508596</td>
<td>406821</td>
<td>185665</td>
<td>176581</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>20201061</td>
<td>5083434</td>
<td>1323368</td>
<td>365086</td>
<td>111199</td>
<td>55613</td>
<td>36051</td>
<td>37040</td>
</tr>
<tr>
<td>XTM</td>
<td>20201061</td>
<td>5080273</td>
<td>1309375</td>
<td>359148</td>
<td>121911</td>
<td>64274</td>
<td>47564</td>
<td>42096</td>
</tr>
<tr>
<td>XTM-C</td>
<td>20201061</td>
<td>5155594</td>
<td>1363204</td>
<td>391722</td>
<td>158337</td>
<td>81674</td>
<td>80557</td>
<td>86459</td>
</tr>
</tbody>
</table>

Table C.4: AQ(0.01) - Running Times (cycles).
### Table C.5: AQ(0.005) – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mrgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>42849690</td>
<td>10853963</td>
<td>2753885</td>
<td>702964</td>
<td>184803</td>
<td>56967</td>
<td>25354</td>
<td>23975</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>43075918</td>
<td>11296466</td>
<td>2856836</td>
<td>723780</td>
<td>191625</td>
<td>62494</td>
<td>30202</td>
<td>25601</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>43062610</td>
<td>10776279</td>
<td>2717726</td>
<td>710372</td>
<td>189768</td>
<td>62878</td>
<td>32043</td>
<td>26622</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>43068273</td>
<td>10781077</td>
<td>2715584</td>
<td>695366</td>
<td>189189</td>
<td>63505</td>
<td>32768</td>
<td>26708</td>
</tr>
<tr>
<td>RR-1</td>
<td>43062592</td>
<td>10780173</td>
<td>2720015</td>
<td>726801</td>
<td>223000</td>
<td>125304</td>
<td>117578</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>43062592</td>
<td>10792555</td>
<td>2728316</td>
<td>723552</td>
<td>210191</td>
<td>104606</td>
<td>105919</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>43832400</td>
<td>11785704</td>
<td>3060293</td>
<td>806422</td>
<td>449619</td>
<td>441495</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>43832400</td>
<td>12127888</td>
<td>3201159</td>
<td>851806</td>
<td>276455</td>
<td>214594</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>43062592</td>
<td>10802275</td>
<td>2763857</td>
<td>746424</td>
<td>214560</td>
<td>84572</td>
<td>51378</td>
<td>41642</td>
</tr>
<tr>
<td>XTM</td>
<td>43062592</td>
<td>10809566</td>
<td>2763583</td>
<td>731563</td>
<td>226962</td>
<td>105396</td>
<td>64571</td>
<td>62185</td>
</tr>
<tr>
<td>XTM-C</td>
<td>43068255</td>
<td>10974037</td>
<td>2877550</td>
<td>800739</td>
<td>312924</td>
<td>141977</td>
<td>100445</td>
<td>131108</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mrgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>212564610</td>
<td>53811875</td>
<td>13609923</td>
<td>3482838</td>
<td>875855</td>
<td>231076</td>
<td>69970</td>
<td>29874</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>213632152</td>
<td>55949997</td>
<td>14130644</td>
<td>3545005</td>
<td>900699</td>
<td>239856</td>
<td>74201</td>
<td>34786</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>213609518</td>
<td>53409304</td>
<td>13379388</td>
<td>3442923</td>
<td>877465</td>
<td>238553</td>
<td>78261</td>
<td>38143</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>213615181</td>
<td>53436659</td>
<td>13389600</td>
<td>3467763</td>
<td>934271</td>
<td>328244</td>
<td>280841</td>
<td>--</td>
</tr>
<tr>
<td>RR-1</td>
<td>213620826</td>
<td>53416489</td>
<td>13385074</td>
<td>3423732</td>
<td>954951</td>
<td>403871</td>
<td>326324</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>213609500</td>
<td>53438267</td>
<td>13406936</td>
<td>3467763</td>
<td>934271</td>
<td>328244</td>
<td>280841</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>217412192</td>
<td>58408211</td>
<td>15120932</td>
<td>3888689</td>
<td>1080120</td>
<td>916236</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>217440993</td>
<td>60112951</td>
<td>15824024</td>
<td>4111792</td>
<td>1094066</td>
<td>640743</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>213632152</td>
<td>53479234</td>
<td>13505753</td>
<td>3549928</td>
<td>950098</td>
<td>301909</td>
<td>118771</td>
<td>63726</td>
</tr>
<tr>
<td>XTM</td>
<td>213626489</td>
<td>53500740</td>
<td>13502898</td>
<td>3448766</td>
<td>932373</td>
<td>345939</td>
<td>157037</td>
<td>96147</td>
</tr>
<tr>
<td>XTM-C</td>
<td>213615163</td>
<td>54176113</td>
<td>14022452</td>
<td>3741867</td>
<td>118771</td>
<td>63726</td>
<td>158011</td>
<td>--</td>
</tr>
</tbody>
</table>

186
### C.1.2 Variable $t_n$

<table>
<thead>
<tr>
<th>$p$ (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mgr.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>20101185</td>
<td>5094293</td>
<td>1293694</td>
<td>33288</td>
<td>93357</td>
<td>31832</td>
<td>20537</td>
<td>20537</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>20212287</td>
<td>5395435</td>
<td>1344670</td>
<td>346903</td>
<td>99621</td>
<td>36910</td>
<td>24814</td>
<td>24776</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>20201097</td>
<td>5059462</td>
<td>1283266</td>
<td>334912</td>
<td>98373</td>
<td>39536</td>
<td>27556</td>
<td>27556</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>20201097</td>
<td>5064444</td>
<td>1279749</td>
<td>333313</td>
<td>97137</td>
<td>41090</td>
<td>27873</td>
<td>27584</td>
</tr>
<tr>
<td>RR-1</td>
<td>20201061</td>
<td>5063218</td>
<td>1303170</td>
<td>383990</td>
<td>153659</td>
<td>99295</td>
<td>97459</td>
<td>97459</td>
</tr>
<tr>
<td>RR-2</td>
<td>20201061</td>
<td>5067324</td>
<td>1327694</td>
<td>367722</td>
<td>140573</td>
<td>104137</td>
<td>97089</td>
<td>97089</td>
</tr>
<tr>
<td>Diff-1</td>
<td>20201061</td>
<td>5873382</td>
<td>1573857</td>
<td>445112</td>
<td>325325</td>
<td>305601</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Diff-2</td>
<td>20201061</td>
<td>6046213</td>
<td>1636998</td>
<td>451022</td>
<td>250021</td>
<td>247527</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TTM</td>
<td>20201061</td>
<td>5092247</td>
<td>1294561</td>
<td>332725</td>
<td>91614</td>
<td>34278</td>
<td>20743</td>
<td>20743</td>
</tr>
<tr>
<td>XTM</td>
<td>20201061</td>
<td>5303552</td>
<td>1345158</td>
<td>349025</td>
<td>98403</td>
<td>38643</td>
<td>25714</td>
<td>25778</td>
</tr>
<tr>
<td>XTM-C</td>
<td>20206724</td>
<td>5157174</td>
<td>159428</td>
<td>72777</td>
<td>81601</td>
<td>127863</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$p$ (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mgr.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>20101211</td>
<td>5092227</td>
<td>1294561</td>
<td>332725</td>
<td>91614</td>
<td>34278</td>
<td>20743</td>
<td>20743</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>20212437</td>
<td>5303939</td>
<td>1345158</td>
<td>349025</td>
<td>98403</td>
<td>38643</td>
<td>25714</td>
<td>25778</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>20212477</td>
<td>5061063</td>
<td>1285444</td>
<td>336478</td>
<td>99937</td>
<td>40093</td>
<td>28199</td>
<td>29359</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>20211511</td>
<td>5066982</td>
<td>1283950</td>
<td>338640</td>
<td>98301</td>
<td>41170</td>
<td>27550</td>
<td>30351</td>
</tr>
<tr>
<td>RR-1</td>
<td>20211113</td>
<td>5066859</td>
<td>1308516</td>
<td>376101</td>
<td>165350</td>
<td>122435</td>
<td>144460</td>
<td>-</td>
</tr>
<tr>
<td>RR-2</td>
<td>20211113</td>
<td>5066859</td>
<td>1308516</td>
<td>376101</td>
<td>165350</td>
<td>122435</td>
<td>144460</td>
<td>-</td>
</tr>
<tr>
<td>Diff-1</td>
<td>20563652</td>
<td>6565060</td>
<td>1629308</td>
<td>528693</td>
<td>352325</td>
<td>377463</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Diff-2</td>
<td>20563652</td>
<td>6732636</td>
<td>1893449</td>
<td>524699</td>
<td>306926</td>
<td>309844</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TTM</td>
<td>20201197</td>
<td>5085276</td>
<td>1302267</td>
<td>382041</td>
<td>226899</td>
<td>188339</td>
<td>196214</td>
<td>-</td>
</tr>
<tr>
<td>XTM</td>
<td>20201235</td>
<td>5073495</td>
<td>1309811</td>
<td>355473</td>
<td>120006</td>
<td>70032</td>
<td>47941</td>
<td>46875</td>
</tr>
<tr>
<td>XTM-C</td>
<td>2021153</td>
<td>5148725</td>
<td>1370264</td>
<td>396630</td>
<td>146105</td>
<td>84630</td>
<td>93370</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$p$ (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mgr.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>20101263</td>
<td>5089159</td>
<td>1296128</td>
<td>336103</td>
<td>92870</td>
<td>33446</td>
<td>21237</td>
<td>21237</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>20206850</td>
<td>5305578</td>
<td>1347103</td>
<td>346523</td>
<td>98524</td>
<td>40262</td>
<td>28304</td>
<td>27134</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>20201223</td>
<td>5060882</td>
<td>1287232</td>
<td>338458</td>
<td>104206</td>
<td>45923</td>
<td>31178</td>
<td>36547</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>20201223</td>
<td>5066227</td>
<td>1287395</td>
<td>339847</td>
<td>104544</td>
<td>43762</td>
<td>31813</td>
<td>38917</td>
</tr>
<tr>
<td>RR-1</td>
<td>20201820</td>
<td>5062136</td>
<td>1321056</td>
<td>439734</td>
<td>226899</td>
<td>188339</td>
<td>196214</td>
<td>-</td>
</tr>
<tr>
<td>RR-2</td>
<td>20211911</td>
<td>5100399</td>
<td>1331959</td>
<td>383770</td>
<td>213222</td>
<td>190526</td>
<td>199654</td>
<td>-</td>
</tr>
<tr>
<td>Diff-1</td>
<td>20562577</td>
<td>7935005</td>
<td>2341666</td>
<td>673266</td>
<td>440028</td>
<td>468642</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Diff-2</td>
<td>20566330</td>
<td>8060183</td>
<td>2389992</td>
<td>693360</td>
<td>478788</td>
<td>442688</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TTM</td>
<td>20206986</td>
<td>5093238</td>
<td>1346358</td>
<td>378601</td>
<td>139401</td>
<td>78497</td>
<td>79104</td>
<td>67661</td>
</tr>
<tr>
<td>XTM</td>
<td>20201343</td>
<td>5076470</td>
<td>1309059</td>
<td>359356</td>
<td>146317</td>
<td>72398</td>
<td>54868</td>
<td>54363</td>
</tr>
<tr>
<td>XTM-C</td>
<td>20201225</td>
<td>5157686</td>
<td>1362555</td>
<td>399678</td>
<td>142857</td>
<td>87429</td>
<td>82477</td>
<td>85041</td>
</tr>
</tbody>
</table>

Table C.9: $AQ(0.01) - t_n = 8$ Cycles / Flit-Hop - Running Times (cycles).
<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>20101367</td>
<td>5087882</td>
<td>1289253</td>
<td>330762</td>
<td>93377</td>
<td>35005</td>
<td>21432</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>20201349</td>
<td>5313573</td>
<td>1349120</td>
<td>348855</td>
<td>100552</td>
<td>40071</td>
<td>28722</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>20201385</td>
<td>5061883</td>
<td>1296083</td>
<td>340302</td>
<td>105081</td>
<td>47603</td>
<td>38895</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>20212711</td>
<td>5073267</td>
<td>1287423</td>
<td>342404</td>
<td>107550</td>
<td>49316</td>
<td>41389</td>
</tr>
<tr>
<td>RR-1</td>
<td>20202285</td>
<td>5094770</td>
<td>1330448</td>
<td>480289</td>
<td>272499</td>
<td>276916</td>
<td>254097</td>
</tr>
<tr>
<td>RR-2</td>
<td>20201347</td>
<td>5129123</td>
<td>1353045</td>
<td>461425</td>
<td>317963</td>
<td>313609</td>
<td>304211</td>
</tr>
<tr>
<td>Diff-1</td>
<td>20562599</td>
<td>10683379</td>
<td>3333567</td>
<td>994517</td>
<td>702582</td>
<td>687026</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>20574141</td>
<td>10838485</td>
<td>3401989</td>
<td>1036899</td>
<td>685123</td>
<td>657468</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>20213124</td>
<td>5110798</td>
<td>1374152</td>
<td>428394</td>
<td>173319</td>
<td>93025</td>
<td>87635</td>
</tr>
<tr>
<td>XTM</td>
<td>20224039</td>
<td>5101166</td>
<td>1323069</td>
<td>382214</td>
<td>180691</td>
<td>94893</td>
<td>84169</td>
</tr>
<tr>
<td>XTM-C</td>
<td>20201387</td>
<td>5162093</td>
<td>1368166</td>
<td>439721</td>
<td>170311</td>
<td>93938</td>
<td>93656</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>20197722</td>
<td>5099148</td>
<td>1292733</td>
<td>335858</td>
<td>95552</td>
<td>36017</td>
<td>26176</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>20300389</td>
<td>5279082</td>
<td>1344262</td>
<td>354152</td>
<td>112299</td>
<td>59733</td>
<td>55886</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>20300533</td>
<td>5098486</td>
<td>1336288</td>
<td>388929</td>
<td>133236</td>
<td>83669</td>
<td>92458</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>20300533</td>
<td>5112646</td>
<td>1318222</td>
<td>382564</td>
<td>144596</td>
<td>90679</td>
<td>115074</td>
</tr>
<tr>
<td>RR-1</td>
<td>20300433</td>
<td>5219740</td>
<td>1599019</td>
<td>808819</td>
<td>634666</td>
<td>634270</td>
<td>615613</td>
</tr>
<tr>
<td>RR-2</td>
<td>20300435</td>
<td>5219832</td>
<td>1550463</td>
<td>808250</td>
<td>593332</td>
<td>658476</td>
<td>683631</td>
</tr>
<tr>
<td>Diff-1</td>
<td>20804163</td>
<td>27373382</td>
<td>9347160</td>
<td>3095301</td>
<td>2713481</td>
<td>2643436</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>20804163</td>
<td>27637925</td>
<td>9500319</td>
<td>3103397</td>
<td>2176886</td>
<td>2129328</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>20301711</td>
<td>5163685</td>
<td>1457252</td>
<td>464483</td>
<td>273988</td>
<td>207599</td>
<td>192060</td>
</tr>
<tr>
<td>XTM</td>
<td>20306500</td>
<td>5209530</td>
<td>1396477</td>
<td>470213</td>
<td>236521</td>
<td>152687</td>
<td>158708</td>
</tr>
<tr>
<td>XTM-C</td>
<td>20300837</td>
<td>5194643</td>
<td>1464410</td>
<td>495743</td>
<td>283201</td>
<td>263703</td>
<td>358852</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>635387</td>
<td>157638</td>
<td>40751</td>
<td>12845</td>
<td>6451</td>
<td>5413</td>
<td>5412</td>
<td>5412</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>652439</td>
<td>181509</td>
<td>50068</td>
<td>16011</td>
<td>9000</td>
<td>7615</td>
<td>7597</td>
<td>7615</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>652457</td>
<td>168317</td>
<td>47300</td>
<td>17914</td>
<td>10660</td>
<td>9565</td>
<td>10007</td>
<td>10601</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>652457</td>
<td>168714</td>
<td>47871</td>
<td>18885</td>
<td>10564</td>
<td>9907</td>
<td>9701</td>
<td>10493</td>
</tr>
<tr>
<td>RR-1</td>
<td>652439</td>
<td>168910</td>
<td>51950</td>
<td>23679</td>
<td>19527</td>
<td>17932</td>
<td>18455</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>652439</td>
<td>169247</td>
<td>51334</td>
<td>23742</td>
<td>19366</td>
<td>20220</td>
<td>20292</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>664725</td>
<td>182007</td>
<td>49471</td>
<td>140830</td>
<td>40976</td>
<td>40976</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>664725</td>
<td>188783</td>
<td>49769</td>
<td>29616</td>
<td>30894</td>
<td>29412</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>652439</td>
<td>176628</td>
<td>56311</td>
<td>25182</td>
<td>18883</td>
<td>19084</td>
<td>17928</td>
<td>--</td>
</tr>
<tr>
<td>XTM</td>
<td>652439</td>
<td>174038</td>
<td>57631</td>
<td>28755</td>
<td>18556</td>
<td>20204</td>
<td>17600</td>
<td>--</td>
</tr>
<tr>
<td>XTM-C</td>
<td>652439</td>
<td>184850</td>
<td>67767</td>
<td>33709</td>
<td>33980</td>
<td>30217</td>
<td>53639</td>
<td>57930</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>7055052</td>
<td>1745573</td>
<td>429066</td>
<td>110201</td>
<td>31113</td>
<td>12109</td>
<td>7712</td>
<td>7353</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>724444</td>
<td>1988825</td>
<td>517631</td>
<td>135793</td>
<td>39242</td>
<td>16469</td>
<td>10926</td>
<td>10432</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>724446</td>
<td>1816564</td>
<td>461306</td>
<td>125929</td>
<td>40277</td>
<td>19130</td>
<td>14381</td>
<td>15353</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>724446</td>
<td>1821342</td>
<td>464670</td>
<td>126670</td>
<td>40558</td>
<td>18665</td>
<td>15266</td>
<td>15659</td>
</tr>
<tr>
<td>RR-1</td>
<td>724444</td>
<td>1818304</td>
<td>471419</td>
<td>142426</td>
<td>62004</td>
<td>52177</td>
<td>51408</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>724444</td>
<td>1820234</td>
<td>484434</td>
<td>147437</td>
<td>73138</td>
<td>58001</td>
<td>54714</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>737362</td>
<td>1981124</td>
<td>526230</td>
<td>160245</td>
<td>119901</td>
<td>112094</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>737362</td>
<td>2044501</td>
<td>552452</td>
<td>164149</td>
<td>94144</td>
<td>92985</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>724444</td>
<td>1837674</td>
<td>494438</td>
<td>146775</td>
<td>62914</td>
<td>40510</td>
<td>29899</td>
<td>29988</td>
</tr>
<tr>
<td>XTM</td>
<td>724444</td>
<td>1845098</td>
<td>487584</td>
<td>155452</td>
<td>76146</td>
<td>44160</td>
<td>30403</td>
<td>34245</td>
</tr>
<tr>
<td>XTM-C</td>
<td>724444</td>
<td>1909502</td>
<td>574192</td>
<td>193382</td>
<td>96755</td>
<td>73132</td>
<td>156931</td>
<td>224980</td>
</tr>
</tbody>
</table>

Table C.13: FIB(20) - Running Times (cycles).
Table C.14: FIB(25) – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>$p$ (number of processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>78250232</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>80350904</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>80350922</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>80350922</td>
</tr>
<tr>
<td>RR-1</td>
<td>80350904</td>
</tr>
<tr>
<td>RR-2</td>
<td>80350904</td>
</tr>
<tr>
<td>Diff-1</td>
<td>81777563</td>
</tr>
<tr>
<td>Diff-2</td>
<td>81777563</td>
</tr>
<tr>
<td>TTM</td>
<td>80350904</td>
</tr>
<tr>
<td>XTM</td>
<td>80350904</td>
</tr>
<tr>
<td>XTM-C</td>
<td>80350004</td>
</tr>
</tbody>
</table>

Table C.14: FIB(25) – Running Times (cycles).
C.2.2 Variable $t_n$

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>7055092</td>
<td>1748341</td>
<td>432185</td>
<td>110991</td>
<td>31265</td>
<td>11900</td>
<td>7559</td>
<td>7546</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>7244444</td>
<td>1985597</td>
<td>518016</td>
<td>135509</td>
<td>40521</td>
<td>17066</td>
<td>11760</td>
<td>11202</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>7244462</td>
<td>1818527</td>
<td>464326</td>
<td>126600</td>
<td>40595</td>
<td>21158</td>
<td>16430</td>
<td>18562</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>7244462</td>
<td>1820591</td>
<td>466076</td>
<td>127340</td>
<td>42450</td>
<td>21794</td>
<td>16587</td>
<td>17976</td>
</tr>
<tr>
<td>RR-1</td>
<td>7244444</td>
<td>1822291</td>
<td>474991</td>
<td>145363</td>
<td>73299</td>
<td>60292</td>
<td>63251</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>7244444</td>
<td>1828478</td>
<td>478125</td>
<td>150398</td>
<td>84345</td>
<td>71391</td>
<td>75580</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>7373688</td>
<td>2107538</td>
<td>572836</td>
<td>176241</td>
<td>148413</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>7373688</td>
<td>2167362</td>
<td>592608</td>
<td>171304</td>
<td>116465</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>7244444</td>
<td>1811330</td>
<td>491068</td>
<td>139389</td>
<td>87070</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>7244444</td>
<td>1867459</td>
<td>494315</td>
<td>151894</td>
<td>60812</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XTM-C</td>
<td>7244444</td>
<td>1961428</td>
<td>597400</td>
<td>206157</td>
<td>178461</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>7055078</td>
<td>1750182</td>
<td>432740</td>
<td>110666</td>
<td>31683</td>
<td>11952</td>
<td>8003</td>
<td>7912</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>7244480</td>
<td>2010998</td>
<td>523937</td>
<td>139389</td>
<td>41342</td>
<td>17864</td>
<td>12827</td>
<td>12977</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>7244498</td>
<td>1819160</td>
<td>465784</td>
<td>133385</td>
<td>47757</td>
<td>26280</td>
<td>26155</td>
<td>33170</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>7244498</td>
<td>1826416</td>
<td>467037</td>
<td>130588</td>
<td>48097</td>
<td>30072</td>
<td>27037</td>
<td>39954</td>
</tr>
<tr>
<td>RR-1</td>
<td>7244470</td>
<td>1824799</td>
<td>474802</td>
<td>152429</td>
<td>87093</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>7244470</td>
<td>1837599</td>
<td>480901</td>
<td>179041</td>
<td>108505</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>7408048</td>
<td>2374267</td>
<td>671046</td>
<td>208648</td>
<td>160240</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>7408048</td>
<td>2424098</td>
<td>693433</td>
<td>205738</td>
<td>130195</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>7244562</td>
<td>1837068</td>
<td>497081</td>
<td>163188</td>
<td>69283</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>7244572</td>
<td>1821505</td>
<td>481610</td>
<td>150663</td>
<td>75886</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XTM-C</td>
<td>7244490</td>
<td>1943942</td>
<td>558931</td>
<td>197515</td>
<td>149228</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>7055130</td>
<td>1780261</td>
<td>433159</td>
<td>111205</td>
<td>31719</td>
<td>13377</td>
<td>8936</td>
<td>8872</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>7244516</td>
<td>2019017</td>
<td>527771</td>
<td>140046</td>
<td>42754</td>
<td>20686</td>
<td>15818</td>
<td>16647</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>7244534</td>
<td>1820232</td>
<td>465783</td>
<td>133385</td>
<td>47757</td>
<td>26280</td>
<td>26155</td>
<td>33170</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>7244534</td>
<td>1826416</td>
<td>467037</td>
<td>130588</td>
<td>48097</td>
<td>30072</td>
<td>27037</td>
<td>39954</td>
</tr>
<tr>
<td>RR-1</td>
<td>7244522</td>
<td>1827965</td>
<td>498670</td>
<td>179041</td>
<td>108505</td>
<td>113288</td>
<td>108732</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>7244522</td>
<td>1850326</td>
<td>508744</td>
<td>210175</td>
<td>171235</td>
<td>165516</td>
<td>157364</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>7373992</td>
<td>2062551</td>
<td>858206</td>
<td>281516</td>
<td>213001</td>
<td>208463</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>7373992</td>
<td>2917805</td>
<td>875363</td>
<td>293488</td>
<td>240687</td>
<td>236879</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>7244652</td>
<td>1848180</td>
<td>520420</td>
<td>181886</td>
<td>80810</td>
<td>65572</td>
<td>78751</td>
<td>70533</td>
</tr>
<tr>
<td>XTM</td>
<td>7244662</td>
<td>1836490</td>
<td>495480</td>
<td>159742</td>
<td>70338</td>
<td>53176</td>
<td>40310</td>
<td>47491</td>
</tr>
<tr>
<td>XTM-C</td>
<td>7244544</td>
<td>1936986</td>
<td>604019</td>
<td>210688</td>
<td>119359</td>
<td>141376</td>
<td>195389</td>
<td>270567</td>
</tr>
</tbody>
</table>

Table C.17: FIB(20) - $t_n = 8$ Cycles / Flit-Hop - Running Times (cycles).
Table C.18: \( FIB(20) - t_n = 16 \) Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>7055234</td>
<td>1798918</td>
<td>452382</td>
<td>452382</td>
<td>117620</td>
<td>34243</td>
<td>14341</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>7244624</td>
<td>1921304</td>
<td>500173</td>
<td>135957</td>
<td>45422</td>
<td>25199</td>
<td>22033</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>7244624</td>
<td>1824620</td>
<td>489412</td>
<td>133447</td>
<td>54192</td>
<td>34348</td>
<td>42492</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>7244624</td>
<td>1836399</td>
<td>475337</td>
<td>138790</td>
<td>56558</td>
<td>41500</td>
<td>56770</td>
</tr>
<tr>
<td>RR-1</td>
<td>7244626</td>
<td>1868163</td>
<td>523030</td>
<td>212372</td>
<td>159375</td>
<td>148379</td>
<td>153555</td>
</tr>
<tr>
<td>RR-2</td>
<td>7244626</td>
<td>1934930</td>
<td>620762</td>
<td>332189</td>
<td>275455</td>
<td>250806</td>
<td>222271</td>
</tr>
<tr>
<td>Diff-1</td>
<td>7373786</td>
<td>3852791</td>
<td>1243938</td>
<td>420987</td>
<td>348716</td>
<td>327626</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>7373786</td>
<td>3902435</td>
<td>1249959</td>
<td>403378</td>
<td>311929</td>
<td>307490</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>7244850</td>
<td>1878020</td>
<td>535304</td>
<td>191020</td>
<td>95326</td>
<td>117295</td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>7244850</td>
<td>1855064</td>
<td>537298</td>
<td>202074</td>
<td>103859</td>
<td>74319</td>
<td>66744</td>
</tr>
<tr>
<td>XTM-C</td>
<td>7244634</td>
<td>1947001</td>
<td>600314</td>
<td>278499</td>
<td>305830</td>
<td>244221</td>
<td>390895</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>7130049</td>
<td>1834009</td>
<td>467888</td>
<td>128719</td>
<td>42126</td>
<td>21906</td>
<td>21585</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>7324486</td>
<td>1865705</td>
<td>499085</td>
<td>156675</td>
<td>82906</td>
<td>88391</td>
<td>77426</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>7316384</td>
<td>1853647</td>
<td>493737</td>
<td>151475</td>
<td>79889</td>
<td>69690</td>
<td>99737</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>7316384</td>
<td>1866739</td>
<td>502636</td>
<td>160024</td>
<td>82932</td>
<td>76475</td>
<td>112934</td>
</tr>
<tr>
<td>RR-1</td>
<td>7324510</td>
<td>1958795</td>
<td>699563</td>
<td>420188</td>
<td>352456</td>
<td>330024</td>
<td>346777</td>
</tr>
<tr>
<td>RR-2</td>
<td>7324510</td>
<td>2015863</td>
<td>688651</td>
<td>44963</td>
<td>47633</td>
<td>422665</td>
<td>435985</td>
</tr>
<tr>
<td>Diff-1</td>
<td>7425235</td>
<td>9997839</td>
<td>3553372</td>
<td>1417313</td>
<td>1256328</td>
<td>1249922</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>7425235</td>
<td>1006936</td>
<td>3524931</td>
<td>1321262</td>
<td>1144075</td>
<td>1132898</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>7325460</td>
<td>1905376</td>
<td>618049</td>
<td>284587</td>
<td>179006</td>
<td>198796</td>
<td>240065</td>
</tr>
<tr>
<td>XTM</td>
<td>7331310</td>
<td>1868209</td>
<td>587810</td>
<td>229439</td>
<td>156843</td>
<td>128660</td>
<td>134495</td>
</tr>
<tr>
<td>XTM-C</td>
<td>7324634</td>
<td>1947001</td>
<td>600314</td>
<td>278499</td>
<td>305830</td>
<td>244221</td>
<td>390895</td>
</tr>
</tbody>
</table>
### C.3 TSP

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

<table>
<thead>
<tr>
<th></th>
<th>Mgr.</th>
<th>$p$ (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Free-Ideal</strong></td>
<td>3514490</td>
<td>543917</td>
<td>144389</td>
<td>51611</td>
<td>16562</td>
<td>11679</td>
<td>11643</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P-Ideal</strong></td>
<td>3635870</td>
<td>631272</td>
<td>157874</td>
<td>47468</td>
<td>20371</td>
<td>14114</td>
<td>17196</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C-Ideal-1</strong></td>
<td>3635888</td>
<td>588285</td>
<td>148815</td>
<td>49261</td>
<td>22223</td>
<td>14651</td>
<td>18501</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C-Ideal-2</strong></td>
<td>3635888</td>
<td>598458</td>
<td>147554</td>
<td>47614</td>
<td>20880</td>
<td>14099</td>
<td>18059</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RR-1</strong></td>
<td>3635870</td>
<td>579760</td>
<td>152304</td>
<td>56602</td>
<td>31083</td>
<td>28385</td>
<td>30616</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RR-2</strong></td>
<td>3635870</td>
<td>590096</td>
<td>154367</td>
<td>55785</td>
<td>31167</td>
<td>29900</td>
<td>32736</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Diff-1</strong></td>
<td>3701646</td>
<td>785756</td>
<td>156168</td>
<td>55840</td>
<td>30369</td>
<td>24169</td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Diff-2</strong></td>
<td>3701646</td>
<td>622469</td>
<td>175249</td>
<td>56602</td>
<td>33108</td>
<td>28385</td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TTM</strong></td>
<td>3635870</td>
<td>785756</td>
<td>156168</td>
<td>55840</td>
<td>30369</td>
<td>24169</td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>XTM</strong></td>
<td>3635870</td>
<td>599244</td>
<td>158474</td>
<td>58582</td>
<td>39826</td>
<td>34023</td>
<td>30325</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>XTM-C</strong></td>
<td>3635870</td>
<td>805614</td>
<td>171855</td>
<td>74057</td>
<td>50162</td>
<td>40302</td>
<td>49346</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Mgr.</th>
<th>$p$ (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Free-Ideal</strong></td>
<td>12188860</td>
<td>2810166</td>
<td>600440</td>
<td>179383</td>
<td>50518</td>
<td>19740</td>
<td>24325</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P-Ideal</strong></td>
<td>12613984</td>
<td>331219</td>
<td>910147</td>
<td>222944</td>
<td>61708</td>
<td>22862</td>
<td>24671</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C-Ideal-1</strong></td>
<td>12614002</td>
<td>327010</td>
<td>867710</td>
<td>218974</td>
<td>60053</td>
<td>2028</td>
<td>25192</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C-Ideal-2</strong></td>
<td>12614002</td>
<td>306803</td>
<td>813595</td>
<td>222760</td>
<td>63100</td>
<td>24771</td>
<td>26176</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RR-1</strong></td>
<td>12613984</td>
<td>325456</td>
<td>877521</td>
<td>237613</td>
<td>89107</td>
<td>64305</td>
<td>61712</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RR-2</strong></td>
<td>12613984</td>
<td>312529</td>
<td>829885</td>
<td>243429</td>
<td>94477</td>
<td>59541</td>
<td>66103</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Diff-1</strong></td>
<td>12839426</td>
<td>3633033</td>
<td>1048051</td>
<td>259083</td>
<td>151099</td>
<td>159196</td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Diff-2</strong></td>
<td>12839426</td>
<td>3189390</td>
<td>966454</td>
<td>265024</td>
<td>13908</td>
<td>135128</td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TTM</strong></td>
<td>12613984</td>
<td>3631367</td>
<td>862041</td>
<td>226416</td>
<td>80278</td>
<td>40875</td>
<td>31815</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>XTM</strong></td>
<td>12613984</td>
<td>3012289</td>
<td>845888</td>
<td>242000</td>
<td>85288</td>
<td>60266</td>
<td>39302</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>XTM-C</strong></td>
<td>12613984</td>
<td>3824205</td>
<td>959277</td>
<td>273300</td>
<td>125548</td>
<td>68458</td>
<td>89273</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.21: TSP(9) – Running Times (cycles).
### Table C.22: TSP(10) – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>40284900</td>
<td>9170942</td>
<td>2153612</td>
<td>547642</td>
<td>142919</td>
<td>42270</td>
<td>17978</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>41708532</td>
<td>10508192</td>
<td>2695097</td>
<td>685780</td>
<td>184424</td>
<td>51549</td>
<td>23096</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>41708550</td>
<td>1019072</td>
<td>2463512</td>
<td>635929</td>
<td>171185</td>
<td>52522</td>
<td>25641</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>41708550</td>
<td>10248470</td>
<td>2457766</td>
<td>627366</td>
<td>167683</td>
<td>54446</td>
<td>26210</td>
</tr>
<tr>
<td>RR-1</td>
<td>41708532</td>
<td>9996882</td>
<td>2532657</td>
<td>667974</td>
<td>243249</td>
<td>141835</td>
<td>135856</td>
</tr>
<tr>
<td>RR-2</td>
<td>41708532</td>
<td>10047895</td>
<td>2547202</td>
<td>651861</td>
<td>222340</td>
<td>109408</td>
<td>113971</td>
</tr>
<tr>
<td>Diff-1</td>
<td>42453008</td>
<td>1075891</td>
<td>2828432</td>
<td>733163</td>
<td>301146</td>
<td>283301</td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>42453008</td>
<td>11890537</td>
<td>2876274</td>
<td>762911</td>
<td>246536</td>
<td>244190</td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>17058532</td>
<td>10359111</td>
<td>2493742</td>
<td>668676</td>
<td>217039</td>
<td>78791</td>
<td>48531</td>
</tr>
<tr>
<td>XTM</td>
<td>41708532</td>
<td>10288864</td>
<td>2489567</td>
<td>661523</td>
<td>218552</td>
<td>113454</td>
<td>78960</td>
</tr>
<tr>
<td>XTM-C</td>
<td>41708532</td>
<td>10510311</td>
<td>2785668</td>
<td>774362</td>
<td>293431</td>
<td>128218</td>
<td>158858</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>31504075</td>
<td>70025896</td>
<td>14156147</td>
<td>3044056</td>
<td>948044</td>
<td>197421</td>
<td>57253</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>32603432</td>
<td>67583229</td>
<td>15875151</td>
<td>3913138</td>
<td>1009189</td>
<td>264798</td>
<td>85029</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>32603430</td>
<td>60800979</td>
<td>1443211</td>
<td>3640721</td>
<td>954195</td>
<td>257512</td>
<td>88977</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>32603430</td>
<td>60790603</td>
<td>14446087</td>
<td>3546869</td>
<td>923410</td>
<td>2198580</td>
<td>87020</td>
</tr>
<tr>
<td>RR-1</td>
<td>32603432</td>
<td>60805479</td>
<td>1448958</td>
<td>3618116</td>
<td>924730</td>
<td>608160</td>
<td>477571</td>
</tr>
<tr>
<td>RR-2</td>
<td>32603432</td>
<td>60805439</td>
<td>14484518</td>
<td>3595915</td>
<td>1006446</td>
<td>415116</td>
<td>325957</td>
</tr>
<tr>
<td>Diff-1</td>
<td>33184858</td>
<td>80648967</td>
<td>15999477</td>
<td>4115066</td>
<td>1102629</td>
<td>798231</td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>33184858</td>
<td>82344330</td>
<td>16588289</td>
<td>4588820</td>
<td>1143817</td>
<td>670868</td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>32603432</td>
<td>75165863</td>
<td>17027508</td>
<td>4171091</td>
<td>1041265</td>
<td>332591</td>
<td>143688</td>
</tr>
<tr>
<td>XTM</td>
<td>32603432</td>
<td>60856918</td>
<td>14520085</td>
<td>3418789</td>
<td>1089472</td>
<td>450306</td>
<td>247843</td>
</tr>
<tr>
<td>XTM-C</td>
<td>32603432</td>
<td>75548463</td>
<td>19239072</td>
<td>4262388</td>
<td>1450355</td>
<td>555329</td>
<td>377919</td>
</tr>
</tbody>
</table>

194
C.3.2 Variable \( t_n \)

<table>
<thead>
<tr>
<th>p (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mgr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>4024906</td>
<td>9175657</td>
<td>2153693</td>
<td>542931</td>
<td>149919</td>
<td>43319</td>
<td>19441</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>41708532</td>
<td>10818194</td>
<td>2823914</td>
<td>686782</td>
<td>177128</td>
<td>53554</td>
<td>22994</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>37268656</td>
<td>8938208</td>
<td>2277964</td>
<td>582686</td>
<td>157152</td>
<td>51602</td>
<td>27831</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>37268656</td>
<td>9183199</td>
<td>2325136</td>
<td>567449</td>
<td>154608</td>
<td>52818</td>
<td>27902</td>
</tr>
<tr>
<td>RR-1</td>
<td>41708532</td>
<td>10277961</td>
<td>2607727</td>
<td>674641</td>
<td>269242</td>
<td>179227</td>
<td>185395</td>
</tr>
<tr>
<td>RR-2</td>
<td>41708532</td>
<td>10286993</td>
<td>2608425</td>
<td>663625</td>
<td>258329</td>
<td>156383</td>
<td>149001</td>
</tr>
<tr>
<td>Diff-1</td>
<td>42452926</td>
<td>11835760</td>
<td>3068563</td>
<td>794801</td>
<td>328435</td>
<td>307961</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>42452926</td>
<td>12295460</td>
<td>3248999</td>
<td>841093</td>
<td>326841</td>
<td>330127</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>41708532</td>
<td>10420442</td>
<td>2597085</td>
<td>702539</td>
<td>218844</td>
<td>91824</td>
<td>51894</td>
</tr>
<tr>
<td>XTM</td>
<td>41708532</td>
<td>10210999</td>
<td>2522599</td>
<td>665090</td>
<td>257242</td>
<td>139872</td>
<td>79813</td>
</tr>
<tr>
<td>XTM-C</td>
<td>41708532</td>
<td>10523375</td>
<td>2799160</td>
<td>782442</td>
<td>299168</td>
<td>141508</td>
<td>128238</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>p (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mgr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>40284978</td>
<td>9030180</td>
<td>2154774</td>
<td>54177</td>
<td>14073</td>
<td>54985</td>
<td>20207</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>37268782</td>
<td>9825104</td>
<td>2622726</td>
<td>629292</td>
<td>166593</td>
<td>50832</td>
<td>25689</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>37268782</td>
<td>8939314</td>
<td>2312564</td>
<td>583627</td>
<td>159735</td>
<td>54946</td>
<td>28877</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>37268782</td>
<td>9176199</td>
<td>2325136</td>
<td>567449</td>
<td>154608</td>
<td>52818</td>
<td>27902</td>
</tr>
<tr>
<td>RR-1</td>
<td>37268782</td>
<td>9206744</td>
<td>2221696</td>
<td>675244</td>
<td>280451</td>
<td>226709</td>
<td>214749</td>
</tr>
<tr>
<td>RR-2</td>
<td>37268782</td>
<td>9213020</td>
<td>2236764</td>
<td>659935</td>
<td>283511</td>
<td>200450</td>
<td>186491</td>
</tr>
<tr>
<td>Diff-1</td>
<td>38004678</td>
<td>12072635</td>
<td>3174203</td>
<td>852180</td>
<td>383371</td>
<td>377469</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>38004678</td>
<td>12072548</td>
<td>3270547</td>
<td>870092</td>
<td>345094</td>
<td>338349</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>37268875</td>
<td>9203473</td>
<td>2454967</td>
<td>660813</td>
<td>203503</td>
<td>91824</td>
<td>51894</td>
</tr>
<tr>
<td>XTM</td>
<td>37268875</td>
<td>9235892</td>
<td>2236277</td>
<td>635671</td>
<td>246553</td>
<td>117606</td>
<td>75326</td>
</tr>
<tr>
<td>XTM-C</td>
<td>37268875</td>
<td>9375229</td>
<td>2538273</td>
<td>717532</td>
<td>284910</td>
<td>128558</td>
<td>189253</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>p (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mgr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>42484978</td>
<td>9211855</td>
<td>2244573</td>
<td>545730</td>
<td>142692</td>
<td>44985</td>
<td>20207</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>37268782</td>
<td>9825104</td>
<td>2622726</td>
<td>629292</td>
<td>166593</td>
<td>50832</td>
<td>25689</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>37268782</td>
<td>8940591</td>
<td>2254033</td>
<td>581546</td>
<td>170192</td>
<td>58026</td>
<td>33841</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>37268782</td>
<td>915668</td>
<td>2337640</td>
<td>571036</td>
<td>170004</td>
<td>57057</td>
<td>33969</td>
</tr>
<tr>
<td>RR-1</td>
<td>37268784</td>
<td>9194751</td>
<td>2358986</td>
<td>710420</td>
<td>338604</td>
<td>326150</td>
<td>312106</td>
</tr>
<tr>
<td>RR-2</td>
<td>37268784</td>
<td>9213020</td>
<td>2326764</td>
<td>733833</td>
<td>346484</td>
<td>334363</td>
<td>257655</td>
</tr>
<tr>
<td>Diff-1</td>
<td>37933850</td>
<td>1496301</td>
<td>4036970</td>
<td>1109184</td>
<td>507669</td>
<td>53573</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>37933850</td>
<td>1475072</td>
<td>4143477</td>
<td>1191087</td>
<td>480872</td>
<td>543859</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>37268918</td>
<td>9219140</td>
<td>2372481</td>
<td>662891</td>
<td>214640</td>
<td>95477</td>
<td>73486</td>
</tr>
<tr>
<td>XTM</td>
<td>37268918</td>
<td>9224555</td>
<td>2349966</td>
<td>616615</td>
<td>220696</td>
<td>96298</td>
<td>82983</td>
</tr>
<tr>
<td>XTM-C</td>
<td>37268830</td>
<td>9365410</td>
<td>2547801</td>
<td>763238</td>
<td>255896</td>
<td>126461</td>
<td>120770</td>
</tr>
</tbody>
</table>

Table C.26: \( TSP(10) - t_n = 8 \) Cycles / Flit-Hop – Running Times (cycles).
<table>
<thead>
<tr>
<th>Mgr.</th>
<th>p (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>38269172</td>
<td>8956924</td>
<td>2230310</td>
<td>343726</td>
<td>143442</td>
<td>43646</td>
<td>121654</td>
<td></td>
</tr>
<tr>
<td>P-Ideal</td>
<td>36262376</td>
<td>9675295</td>
<td>2463593</td>
<td>622066</td>
<td>164949</td>
<td>51854</td>
<td>28741</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>36262430</td>
<td>8941216</td>
<td>2264960</td>
<td>594370</td>
<td>168014</td>
<td>61959</td>
<td>41406</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>36262430</td>
<td>9169046</td>
<td>2288068</td>
<td>578868</td>
<td>170942</td>
<td>64228</td>
<td>42374</td>
<td></td>
</tr>
<tr>
<td>RR-1</td>
<td>36262370</td>
<td>8953442</td>
<td>2298015</td>
<td>708163</td>
<td>433603</td>
<td>387426</td>
<td>401201</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>36262370</td>
<td>9196652</td>
<td>2382659</td>
<td>823730</td>
<td>501444</td>
<td>43646</td>
<td>21654</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>35899341</td>
<td>18241108</td>
<td>5501049</td>
<td>1601684</td>
<td>879497</td>
<td>794900</td>
<td>454885</td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>35999341</td>
<td>19612318</td>
<td>6209007</td>
<td>1773702</td>
<td>851644</td>
<td>885747</td>
<td>454885</td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>356262602</td>
<td>9038395</td>
<td>2313249</td>
<td>697930</td>
<td>287532</td>
<td>128701</td>
<td>102346</td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>356262602</td>
<td>9288592</td>
<td>2383363</td>
<td>627707</td>
<td>236523</td>
<td>123597</td>
<td>87116</td>
<td></td>
</tr>
<tr>
<td>XTM-C</td>
<td>3562442</td>
<td>9143246</td>
<td>2495478</td>
<td>773513</td>
<td>285475</td>
<td>141128</td>
<td>117284</td>
<td></td>
</tr>
</tbody>
</table>

Table C.27: TSP(10) - $t_n = 16$ Cycles / Flit-Hop - Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>p (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>35747304</td>
<td>8661550</td>
<td>2157590</td>
<td>518554</td>
<td>146707</td>
<td>49450</td>
<td>27169</td>
<td></td>
</tr>
<tr>
<td>P-Ideal</td>
<td>35125384</td>
<td>9015423</td>
<td>2449341</td>
<td>642126</td>
<td>174718</td>
<td>67864</td>
<td>45381</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>35139827</td>
<td>8438390</td>
<td>2233736</td>
<td>600855</td>
<td>192904</td>
<td>101322</td>
<td>89819</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>35139827</td>
<td>8743088</td>
<td>2248134</td>
<td>619288</td>
<td>201183</td>
<td>120543</td>
<td>191787</td>
<td></td>
</tr>
<tr>
<td>RR-1</td>
<td>35123582</td>
<td>8802746</td>
<td>2393838</td>
<td>987733</td>
<td>693296</td>
<td>679559</td>
<td>673072</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>35123582</td>
<td>9071328</td>
<td>3103631</td>
<td>1525561</td>
<td>1410619</td>
<td>1693728</td>
<td>983679</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>35712861</td>
<td>44713664</td>
<td>1625811</td>
<td>487927</td>
<td>2499625</td>
<td>2493885</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>35712861</td>
<td>46952858</td>
<td>15475347</td>
<td>4958444</td>
<td>2593371</td>
<td>2733879</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>35126186</td>
<td>908347</td>
<td>2505409</td>
<td>866710</td>
<td>423920</td>
<td>260616</td>
<td>336035</td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>35126186</td>
<td>9027484</td>
<td>2530795</td>
<td>789966</td>
<td>324328</td>
<td>262660</td>
<td>225875</td>
<td></td>
</tr>
<tr>
<td>XTM-C</td>
<td>35125382</td>
<td>9169307</td>
<td>2645049</td>
<td>881426</td>
<td>4519001</td>
<td>378891</td>
<td>386189</td>
<td></td>
</tr>
</tbody>
</table>

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

$$t, = \frac{1}{\text{Cycle Flit-Hop}}$$

<table>
<thead>
<tr>
<th>$p$ (number of processors)</th>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>601423</td>
<td>150552</td>
<td>37856</td>
<td>9712</td>
<td>2794</td>
<td>1193</td>
<td>1061</td>
<td>1041</td>
<td></td>
</tr>
<tr>
<td>Stat</td>
<td>601451</td>
<td>151116</td>
<td>38503</td>
<td>10513</td>
<td>3837</td>
<td>2745</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>P-Ideal</td>
<td>601451</td>
<td>172279</td>
<td>43975</td>
<td>11901</td>
<td>11143</td>
<td>11143</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>601469</td>
<td>252194</td>
<td>83453</td>
<td>48228</td>
<td>48236</td>
<td>48179</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>601469</td>
<td>159332</td>
<td>43975</td>
<td>9991</td>
<td>9964</td>
<td>9964</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>RR-1</td>
<td>601451</td>
<td>219990</td>
<td>81157</td>
<td>48730</td>
<td>52629</td>
<td>60438</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>601451</td>
<td>154008</td>
<td>41001</td>
<td>13621</td>
<td>11901</td>
<td>11143</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>612669</td>
<td>165053</td>
<td>49035</td>
<td>42096</td>
<td>41461</td>
<td>41414</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>612669</td>
<td>174999</td>
<td>48831</td>
<td>38377</td>
<td>38377</td>
<td>38381</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>601451</td>
<td>153845</td>
<td>41067</td>
<td>13657</td>
<td>7987</td>
<td>7969</td>
<td>7567</td>
<td>7978</td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>601451</td>
<td>152570</td>
<td>45179</td>
<td>15052</td>
<td>8778</td>
<td>7120</td>
<td>7402</td>
<td>6769</td>
<td></td>
</tr>
<tr>
<td>XTM-C</td>
<td>601451</td>
<td>153482</td>
<td>42785</td>
<td>18586</td>
<td>12128</td>
<td>12555</td>
<td>12853</td>
<td>13973</td>
<td></td>
</tr>
</tbody>
</table>

Table C.29: UNBAL(1024) - Running Times (cycles).

<table>
<thead>
<tr>
<th>$p$ (number of processors)</th>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>2404687</td>
<td>601368</td>
<td>150560</td>
<td>37888</td>
<td>9838</td>
<td>2954</td>
<td>1337</td>
<td>1213</td>
<td></td>
</tr>
<tr>
<td>Stat</td>
<td>2404715</td>
<td>601932</td>
<td>151207</td>
<td>40689</td>
<td>10881</td>
<td>4573</td>
<td>4243</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>P-Ideal</td>
<td>2404715</td>
<td>683383</td>
<td>173725</td>
<td>44712</td>
<td>41863</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>2404733</td>
<td>1005902</td>
<td>331239</td>
<td>189530</td>
<td>189540</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>2404733</td>
<td>601748</td>
<td>153127</td>
<td>41090</td>
<td>12241</td>
<td>5135</td>
<td>3420</td>
<td>4158</td>
<td></td>
</tr>
<tr>
<td>RR-1</td>
<td>2404715</td>
<td>876055</td>
<td>322925</td>
<td>190020</td>
<td>193884</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>2404715</td>
<td>602224</td>
<td>155512</td>
<td>44082</td>
<td>19137</td>
<td>19393</td>
<td>19327</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>2448707</td>
<td>655806</td>
<td>175443</td>
<td>82893</td>
<td>82893</td>
<td>82893</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>2448707</td>
<td>691825</td>
<td>183335</td>
<td>77300</td>
<td>79073</td>
<td>78355</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>2404715</td>
<td>604955</td>
<td>154140</td>
<td>44036</td>
<td>17450</td>
<td>10850</td>
<td>10214</td>
<td>10680</td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>2404715</td>
<td>603999</td>
<td>163837</td>
<td>50501</td>
<td>23673</td>
<td>12772</td>
<td>11159</td>
<td>10280</td>
<td></td>
</tr>
<tr>
<td>XTM-C</td>
<td>2404715</td>
<td>604513</td>
<td>158667</td>
<td>54386</td>
<td>30082</td>
<td>19281</td>
<td>17258</td>
<td>17765</td>
<td></td>
</tr>
</tbody>
</table>

Table C.30: UNBAL(4096) - Running Times (cycles).
Table C.31: \textit{UNBAL(16384)} – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>9617743</td>
<td>2401632</td>
<td>601376</td>
<td>150592</td>
<td>38014</td>
<td>9998</td>
<td>3098</td>
<td>1466</td>
</tr>
<tr>
<td>Stat</td>
<td>9617717</td>
<td>2401596</td>
<td>602023</td>
<td>151393</td>
<td>39057</td>
<td>11617</td>
<td>6045</td>
<td>7343</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>9617771</td>
<td>2752767</td>
<td>694743</td>
<td>175652</td>
<td>164761</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>9617789</td>
<td>4021080</td>
<td>1321857</td>
<td>754685</td>
<td>754703</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>9617789</td>
<td>2405012</td>
<td>604920</td>
<td>156373</td>
<td>42418</td>
<td>12682</td>
<td>5666</td>
<td>5202</td>
</tr>
<tr>
<td>RR-1</td>
<td>9617771</td>
<td>3504936</td>
<td>1287644</td>
<td>755413</td>
<td>759764</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>9617771</td>
<td>2405488</td>
<td>607116</td>
<td>159412</td>
<td>50449</td>
<td>45451</td>
<td>45390</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>9790833</td>
<td>26�3884</td>
<td>680367</td>
<td>194927</td>
<td>165819</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>9790833</td>
<td>3747652</td>
<td>722464</td>
<td>192409</td>
<td>158327</td>
<td>158215</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>9617771</td>
<td>2408675</td>
<td>603947</td>
<td>156720</td>
<td>44180</td>
<td>18919</td>
<td>13928</td>
<td>14276</td>
</tr>
<tr>
<td>XTM</td>
<td>9617771</td>
<td>2407272</td>
<td>653619</td>
<td>176066</td>
<td>60498</td>
<td>32323</td>
<td>17090</td>
<td>14391</td>
</tr>
<tr>
<td>XTM-C</td>
<td>9617771</td>
<td>2408467</td>
<td>611832</td>
<td>203270</td>
<td>77201</td>
<td>40920</td>
<td>28618</td>
<td>26859</td>
</tr>
</tbody>
</table>

Table C.32: \textit{UNBAL(65536)} – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>38469967</td>
<td>9617688</td>
<td>2404640</td>
<td>601408</td>
<td>150718</td>
<td>30174</td>
<td>10142</td>
<td>3227</td>
</tr>
<tr>
<td>Stat</td>
<td>38469995</td>
<td>9618252</td>
<td>2405287</td>
<td>602209</td>
<td>151761</td>
<td>39793</td>
<td>13089</td>
<td>9265</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>38469995</td>
<td>11010295</td>
<td>2772303</td>
<td>699430</td>
<td>656281</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>384700013</td>
<td>16082583</td>
<td>5284667</td>
<td>3015695</td>
<td>3015695</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>384700013</td>
<td>9618068</td>
<td>2409301</td>
<td>609976</td>
<td>160415</td>
<td>43098</td>
<td>13519</td>
<td>6791</td>
</tr>
<tr>
<td>RR-1</td>
<td>38469995</td>
<td>14013649</td>
<td>5144331</td>
<td>3016122</td>
<td>3020315</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>38469995</td>
<td>2405488</td>
<td>607116</td>
<td>159412</td>
<td>50449</td>
<td>45451</td>
<td>45390</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>39159316</td>
<td>10443171</td>
<td>2699916</td>
<td>705746</td>
<td>326855</td>
<td>327737</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>39159316</td>
<td>2408675</td>
<td>603947</td>
<td>156720</td>
<td>44180</td>
<td>18919</td>
<td>13928</td>
<td>14276</td>
</tr>
<tr>
<td>TTM</td>
<td>38469995</td>
<td>2407272</td>
<td>653619</td>
<td>176066</td>
<td>60498</td>
<td>32323</td>
<td>17090</td>
<td>14391</td>
</tr>
<tr>
<td>XTM</td>
<td>38469995</td>
<td>9620153</td>
<td>2408467</td>
<td>611832</td>
<td>203270</td>
<td>77201</td>
<td>40920</td>
<td>13089</td>
</tr>
<tr>
<td>XTM-C</td>
<td>38469995</td>
<td>9621468</td>
<td>2402075</td>
<td>679842</td>
<td>225614</td>
<td>106160</td>
<td>51351</td>
<td>42882</td>
</tr>
</tbody>
</table>
C.4.2 Variable \( t_n \)

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>9617743</td>
<td>2404639</td>
<td>601377</td>
<td>150587</td>
<td>38047</td>
<td>10027</td>
<td>3101</td>
<td>1430</td>
</tr>
<tr>
<td>Stat</td>
<td>9617771</td>
<td>2405219</td>
<td>602051</td>
<td>151393</td>
<td>39057</td>
<td>11617</td>
<td>6347</td>
<td>7628</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>9617771</td>
<td>2752791</td>
<td>693761</td>
<td>175705</td>
<td>164779</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>9617789</td>
<td>4071548</td>
<td>1361492</td>
<td>754675</td>
<td>754783</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>9617789</td>
<td>2405040</td>
<td>606716</td>
<td>160359</td>
<td>42102</td>
<td>13424</td>
<td>6439</td>
<td>6848</td>
</tr>
<tr>
<td>RR-1</td>
<td>9617771</td>
<td>3553078</td>
<td>1448347</td>
<td>806942</td>
<td>757143</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>9617771</td>
<td>2405541</td>
<td>607736</td>
<td>157020</td>
<td>50633</td>
<td>50529</td>
<td>51265</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>9790833</td>
<td>2776406</td>
<td>741372</td>
<td>215166</td>
<td>184825</td>
<td>184825</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>9790833</td>
<td>2929302</td>
<td>779338</td>
<td>212838</td>
<td>176175</td>
<td>177217</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>9617771</td>
<td>2409171</td>
<td>611465</td>
<td>158732</td>
<td>48466</td>
<td>19116</td>
<td>15863</td>
<td>15170</td>
</tr>
<tr>
<td>XTM</td>
<td>9617771</td>
<td>2406975</td>
<td>612818</td>
<td>176934</td>
<td>60980</td>
<td>30603</td>
<td>18385</td>
<td>14383</td>
</tr>
<tr>
<td>XTM-C</td>
<td>9617771</td>
<td>2408469</td>
<td>618841</td>
<td>183654</td>
<td>72119</td>
<td>36257</td>
<td>30331</td>
<td>30669</td>
</tr>
</tbody>
</table>

Table C.33: UNBAL(16384) – \( t_n = 2 \) Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>9617769</td>
<td>2404671</td>
<td>601403</td>
<td>150587</td>
<td>37997</td>
<td>9935</td>
<td>2991</td>
<td>1330</td>
</tr>
<tr>
<td>Stat</td>
<td>9617797</td>
<td>2405223</td>
<td>602051</td>
<td>151419</td>
<td>39203</td>
<td>11997</td>
<td>6919</td>
<td>8746</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>9617807</td>
<td>2752863</td>
<td>693851</td>
<td>175741</td>
<td>164887</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>9617825</td>
<td>4170649</td>
<td>1421874</td>
<td>754783</td>
<td>754847</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>9617825</td>
<td>2405126</td>
<td>607960</td>
<td>159723</td>
<td>44227</td>
<td>14748</td>
<td>8883</td>
<td>11462</td>
</tr>
<tr>
<td>RR-1</td>
<td>9617797</td>
<td>3924496</td>
<td>1820303</td>
<td>1092005</td>
<td>814483</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>RR-2</td>
<td>9617797</td>
<td>2405541</td>
<td>607736</td>
<td>157020</td>
<td>50633</td>
<td>50529</td>
<td>51265</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>9790833</td>
<td>3101269</td>
<td>862372</td>
<td>255089</td>
<td>221957</td>
<td>221973</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>9790833</td>
<td>3207966</td>
<td>897689</td>
<td>253329</td>
<td>212636</td>
<td>212614</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>9617889</td>
<td>2405702</td>
<td>611702</td>
<td>156291</td>
<td>48826</td>
<td>23947</td>
<td>18998</td>
<td>21534</td>
</tr>
<tr>
<td>XTM</td>
<td>9617899</td>
<td>2407064</td>
<td>612264</td>
<td>171181</td>
<td>70299</td>
<td>30271</td>
<td>19312</td>
<td>16630</td>
</tr>
<tr>
<td>XTM-C</td>
<td>9617817</td>
<td>2410994</td>
<td>621515</td>
<td>194880</td>
<td>66717</td>
<td>35873</td>
<td>34593</td>
<td>30769</td>
</tr>
</tbody>
</table>

Table C.34: UNBAL(16384) – \( t_n = 4 \) Cycles / Flit-Hop – Running Times (cycles).
### Table C.35: UNBAL(16384) – $t_n = 8$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>9617821</td>
<td>2040715</td>
<td>601447</td>
<td>150631</td>
<td>37395</td>
<td>9873</td>
<td>2927</td>
<td>1266</td>
</tr>
<tr>
<td>Stat</td>
<td>9617875</td>
<td>2045299</td>
<td>602171</td>
<td>151669</td>
<td>39593</td>
<td>12673</td>
<td>8115</td>
<td>11008</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>9617861</td>
<td>2752969</td>
<td>694029</td>
<td>175959</td>
<td>165049</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>9618979</td>
<td>4365593</td>
<td>1548553</td>
<td>754959</td>
<td>755143</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>9617897</td>
<td>2405416</td>
<td>606438</td>
<td>159604</td>
<td>45425</td>
<td>17084</td>
<td>12060</td>
<td>17379</td>
</tr>
<tr>
<td>RR-1</td>
<td>9617875</td>
<td>4513450</td>
<td>2346622</td>
<td>1616026</td>
<td>1256941</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>9790841</td>
<td>3752572</td>
<td>1101869</td>
<td>334303</td>
<td>293799</td>
<td>293830</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>9790841</td>
<td>3848858</td>
<td>1136623</td>
<td>331840</td>
<td>281839</td>
<td>282227</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>9617861</td>
<td>2406931</td>
<td>607667</td>
<td>162251</td>
<td>85907</td>
<td>84961</td>
<td>81008</td>
<td>11008</td>
</tr>
<tr>
<td>XTM</td>
<td>9618071</td>
<td>2412800</td>
<td>609717</td>
<td>162150</td>
<td>57617</td>
<td>27659</td>
<td>27477</td>
<td>26717</td>
</tr>
<tr>
<td>XTM-C</td>
<td>9617953</td>
<td>2410835</td>
<td>626021</td>
<td>192675</td>
<td>755411</td>
<td>755377</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

### Table C.36: UNBAL(16384) – $t_n = 16$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>9617925</td>
<td>2040825</td>
<td>601555</td>
<td>150735</td>
<td>38031</td>
<td>9871</td>
<td>29227</td>
<td>--</td>
</tr>
<tr>
<td>Stat</td>
<td>9618031</td>
<td>2045769</td>
<td>602613</td>
<td>152267</td>
<td>40425</td>
<td>13999</td>
<td>8447</td>
<td>113940</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>9618023</td>
<td>2753261</td>
<td>694249</td>
<td>176581</td>
<td>165427</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>9618059</td>
<td>4723315</td>
<td>1791979</td>
<td>755377</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>9618059</td>
<td>2407902</td>
<td>614168</td>
<td>169452</td>
<td>40425</td>
<td>13999</td>
<td>8447</td>
<td>113940</td>
</tr>
<tr>
<td>RR-1</td>
<td>9618031</td>
<td>5386174</td>
<td>3382293</td>
<td>2456214</td>
<td>2004639</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>9790834</td>
<td>5056912</td>
<td>1578660</td>
<td>491322</td>
<td>436759</td>
<td>434702</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>9790834</td>
<td>5177838</td>
<td>1615852</td>
<td>487891</td>
<td>417133</td>
<td>417133</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>9618059</td>
<td>2413018</td>
<td>607866</td>
<td>172037</td>
<td>59377</td>
<td>42768</td>
<td>36099</td>
<td>--</td>
</tr>
<tr>
<td>XTM</td>
<td>9618039</td>
<td>2410285</td>
<td>627809</td>
<td>193619</td>
<td>113314</td>
<td>113940</td>
<td>113940</td>
<td>--</td>
</tr>
<tr>
<td>XTM-C</td>
<td>9618039</td>
<td>2410051</td>
<td>629739</td>
<td>193416</td>
<td>66044</td>
<td>44412</td>
<td>39689</td>
<td>--</td>
</tr>
</tbody>
</table>

### Table C.37: UNBAL(16384) – $t_n = 64$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>9618497</td>
<td>2405385</td>
<td>602123</td>
<td>151307</td>
<td>38031</td>
<td>10427</td>
<td>3383</td>
</tr>
<tr>
<td>Stat</td>
<td>9618993</td>
<td>2407995</td>
<td>605315</td>
<td>155699</td>
<td>45443</td>
<td>22039</td>
<td>23033</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>9618995</td>
<td>2756465</td>
<td>696277</td>
<td>179861</td>
<td>168321</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>9619031</td>
<td>6141369</td>
<td>2867639</td>
<td>1086767</td>
<td>750677</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>9619031</td>
<td>2409000</td>
<td>620783</td>
<td>187299</td>
<td>71467</td>
<td>50757</td>
<td>74059</td>
</tr>
<tr>
<td>RR-1</td>
<td>9618993</td>
<td>7516640</td>
<td>6079406</td>
<td>5169356</td>
<td>4650767</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-1</td>
<td>9791855</td>
<td>12874161</td>
<td>4393430</td>
<td>1438382</td>
<td>1273437</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Diff-2</td>
<td>9791855</td>
<td>13157159</td>
<td>4471812</td>
<td>1418808</td>
<td>1225454</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TTM</td>
<td>9619031</td>
<td>2408285</td>
<td>627809</td>
<td>193619</td>
<td>81576</td>
<td>39756</td>
<td>37971</td>
</tr>
<tr>
<td>XTM</td>
<td>9619031</td>
<td>2400051</td>
<td>629739</td>
<td>193416</td>
<td>66044</td>
<td>44412</td>
<td>39689</td>
</tr>
<tr>
<td>XTM-C</td>
<td>9619031</td>
<td>2426418</td>
<td>640117</td>
<td>219719</td>
<td>107551</td>
<td>89765</td>
<td>95154</td>
</tr>
</tbody>
</table>
C.5 MATMUL: Coarse, Cached

C.5.1 $t_n = 1$ Cycle / Flit-Hop

<table>
<thead>
<tr>
<th></th>
<th>p (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td></td>
<td>202748</td>
<td>85405</td>
<td>30022</td>
<td>16537</td>
</tr>
<tr>
<td>Stat</td>
<td></td>
<td>202748</td>
<td>73246</td>
<td>26632</td>
<td>14552</td>
</tr>
<tr>
<td>P-Ideal</td>
<td></td>
<td>202748</td>
<td>155980</td>
<td>52801</td>
<td>26328</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td></td>
<td>202766</td>
<td>85546</td>
<td>32618</td>
<td>29992</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td></td>
<td>202766</td>
<td>85420</td>
<td>31749</td>
<td>30742</td>
</tr>
<tr>
<td>RR-1</td>
<td></td>
<td>202748</td>
<td>86639</td>
<td>31723</td>
<td>23556</td>
</tr>
<tr>
<td>RR-2</td>
<td></td>
<td>202748</td>
<td>86639</td>
<td>31731</td>
<td>23916</td>
</tr>
<tr>
<td>Diff-1</td>
<td></td>
<td>207312</td>
<td>302232</td>
<td>106888</td>
<td>92857</td>
</tr>
<tr>
<td>Diff-2</td>
<td></td>
<td>207312</td>
<td>94855</td>
<td>31749</td>
<td>30742</td>
</tr>
<tr>
<td>TTM</td>
<td></td>
<td>202748</td>
<td>86551</td>
<td>31932</td>
<td>21687</td>
</tr>
<tr>
<td>XTM</td>
<td></td>
<td>202748</td>
<td>86311</td>
<td>35270</td>
<td>22454</td>
</tr>
</tbody>
</table>

Table C.38: MATMUL(16) (coarse, cached) – Running Times (cycles).

<table>
<thead>
<tr>
<th></th>
<th>p (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td></td>
<td>1596780</td>
<td>522831</td>
<td>151081</td>
<td>55333</td>
<td>33011</td>
</tr>
<tr>
<td>Stat</td>
<td></td>
<td>1596780</td>
<td>475054</td>
<td>139342</td>
<td>49758</td>
<td>29680</td>
</tr>
<tr>
<td>P-Ideal</td>
<td></td>
<td>1596780</td>
<td>996878</td>
<td>286914</td>
<td>99288</td>
<td>52622</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td></td>
<td>1596798</td>
<td>519578</td>
<td>154498</td>
<td>106256</td>
<td>38648</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td></td>
<td>1596798</td>
<td>519452</td>
<td>153179</td>
<td>110777</td>
<td>40071</td>
</tr>
<tr>
<td>RR-1</td>
<td></td>
<td>1596780</td>
<td>62139</td>
<td>153245</td>
<td>62641</td>
<td>62336</td>
</tr>
<tr>
<td>RR-2</td>
<td></td>
<td>1596780</td>
<td>524139</td>
<td>153371</td>
<td>61945</td>
<td>67559</td>
</tr>
<tr>
<td>Diff-1</td>
<td></td>
<td>1625990</td>
<td>2042920</td>
<td>609992</td>
<td>383851</td>
<td>324903</td>
</tr>
<tr>
<td>Diff-2</td>
<td></td>
<td>1625990</td>
<td>571057</td>
<td>315347</td>
<td>196457</td>
<td>144490</td>
</tr>
<tr>
<td>TTM</td>
<td></td>
<td>1596780</td>
<td>524581</td>
<td>153364</td>
<td>59810</td>
<td>42205</td>
</tr>
<tr>
<td>XTM</td>
<td></td>
<td>1596780</td>
<td>523784</td>
<td>153964</td>
<td>61025</td>
<td>46732</td>
</tr>
</tbody>
</table>

Table C.39: MATMUL(32) (coarse, cached) – Running Times (cycles).
<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>12677708</td>
<td>3642945</td>
<td>975307</td>
<td>293114</td>
<td>115611</td>
<td>78707</td>
</tr>
<tr>
<td>Stat</td>
<td>12677708</td>
<td>3454414</td>
<td>930719</td>
<td>272724</td>
<td>101960</td>
<td>68808</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>12677708</td>
<td>7103884</td>
<td>1900267</td>
<td>559296</td>
<td>212970</td>
<td>127776</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>12677726</td>
<td>3629061</td>
<td>981311</td>
<td>574234</td>
<td>123416</td>
<td>91185</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>12677726</td>
<td>3626236</td>
<td>978331</td>
<td>587114</td>
<td>127164</td>
<td>94167</td>
</tr>
<tr>
<td>RR-1</td>
<td>12677708</td>
<td>3644334</td>
<td>979545</td>
<td>301534</td>
<td>154241</td>
<td>189835</td>
</tr>
<tr>
<td>RR-2</td>
<td>12677708</td>
<td>3644334</td>
<td>980529</td>
<td>301534</td>
<td>178596</td>
<td>189722</td>
</tr>
<tr>
<td>Diff-1</td>
<td>12905516</td>
<td>14956668</td>
<td>4150412</td>
<td>2410001</td>
<td>1690097</td>
<td>1434342</td>
</tr>
<tr>
<td>Diff-2</td>
<td>12905516</td>
<td>3962996</td>
<td>2103620</td>
<td>983793</td>
<td>527595</td>
<td>490766</td>
</tr>
<tr>
<td>TTM</td>
<td>12677708</td>
<td>3647009</td>
<td>979384</td>
<td>301944</td>
<td>129253</td>
<td>98591</td>
</tr>
<tr>
<td>XTM</td>
<td>12677708</td>
<td>3643966</td>
<td>980768</td>
<td>302552</td>
<td>132229</td>
<td>109426</td>
</tr>
</tbody>
</table>

Table C.40: MATMUL(64) (coarse, cached) – Running Times (cycles).
C.5.2 Variable $t_n$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>126777906</td>
<td>1361143</td>
<td>269279</td>
<td>360095</td>
<td>429565</td>
<td>245655</td>
</tr>
<tr>
<td>Stat</td>
<td>126777708</td>
<td>1212061</td>
<td>612327</td>
<td>183618</td>
<td>163828</td>
<td>163828</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>126777708</td>
<td>2572751</td>
<td>621327</td>
<td>162327</td>
<td>821318</td>
<td>821318</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>126777708</td>
<td>4663054</td>
<td>1932253</td>
<td>951210</td>
<td>298032</td>
<td>298032</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>126777708</td>
<td>5961410</td>
<td>1918050</td>
<td>1401302</td>
<td>802316</td>
<td>802316</td>
</tr>
<tr>
<td>RR-1</td>
<td>126777708</td>
<td>4733047</td>
<td>1373995</td>
<td>852106</td>
<td>816666</td>
<td>816666</td>
</tr>
<tr>
<td>RR-2</td>
<td>126777708</td>
<td>4733047</td>
<td>1373995</td>
<td>852106</td>
<td>816666</td>
<td>816666</td>
</tr>
<tr>
<td>Diff-1</td>
<td>129055116</td>
<td>20843250</td>
<td>6875081</td>
<td>4379999</td>
<td>3537360</td>
<td>3537360</td>
</tr>
<tr>
<td>Diff-2</td>
<td>129055116</td>
<td>6086145</td>
<td>3557104</td>
<td>1784404</td>
<td>1402990</td>
<td>1402990</td>
</tr>
<tr>
<td>TTM</td>
<td>126777708</td>
<td>4662978</td>
<td>1390074</td>
<td>532596</td>
<td>283898</td>
<td>283898</td>
</tr>
</tbody>
</table>

Table C.41: MATMUL(64) (coarse, cached) – $t_n = 2$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>12677734</td>
<td>4671729</td>
<td>1361143</td>
<td>500790</td>
<td>269279</td>
<td>245655</td>
</tr>
<tr>
<td>Stat</td>
<td>126777708</td>
<td>4077887</td>
<td>1212061</td>
<td>622191</td>
<td>352013</td>
<td>357029</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>126777708</td>
<td>8820271</td>
<td>1623274</td>
<td>821318</td>
<td>623514</td>
<td>623514</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>126777708</td>
<td>4663054</td>
<td>1932253</td>
<td>836134</td>
<td>527621</td>
<td>527621</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>126777708</td>
<td>5961410</td>
<td>1918050</td>
<td>1401302</td>
<td>802316</td>
<td>802316</td>
</tr>
<tr>
<td>RR-1</td>
<td>126777708</td>
<td>4733047</td>
<td>1373995</td>
<td>852106</td>
<td>816666</td>
<td>816666</td>
</tr>
<tr>
<td>RR-2</td>
<td>126777708</td>
<td>4733047</td>
<td>1373995</td>
<td>852106</td>
<td>816666</td>
<td>816666</td>
</tr>
<tr>
<td>Diff-1</td>
<td>129055222</td>
<td>20843250</td>
<td>6875081</td>
<td>4379999</td>
<td>3537360</td>
<td>3537360</td>
</tr>
<tr>
<td>Diff-2</td>
<td>129055222</td>
<td>6086145</td>
<td>3557104</td>
<td>1784404</td>
<td>1402990</td>
<td>1402990</td>
</tr>
<tr>
<td>TTM</td>
<td>126777708</td>
<td>4662978</td>
<td>1390074</td>
<td>532596</td>
<td>283898</td>
<td>283898</td>
</tr>
</tbody>
</table>

Table C.42: MATMUL(64) (coarse, cached) – $t_n = 4$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>12677786</td>
<td>5976757</td>
<td>1885091</td>
<td>767245</td>
<td>473356</td>
<td>489364</td>
</tr>
<tr>
<td>Stat</td>
<td>126777708</td>
<td>4909247</td>
<td>1573680</td>
<td>622191</td>
<td>352013</td>
<td>357029</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>126777708</td>
<td>19367269</td>
<td>6617909</td>
<td>4626254</td>
<td>2491020</td>
<td>2045932</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>126777708</td>
<td>6045105</td>
<td>1932253</td>
<td>836134</td>
<td>527621</td>
<td>527621</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>126777708</td>
<td>5961410</td>
<td>1918050</td>
<td>1401302</td>
<td>802316</td>
<td>802316</td>
</tr>
<tr>
<td>RR-1</td>
<td>126777708</td>
<td>6181500</td>
<td>1903260</td>
<td>839083</td>
<td>673275</td>
<td>107666</td>
</tr>
<tr>
<td>RR-2</td>
<td>126777708</td>
<td>6181500</td>
<td>1918744</td>
<td>852106</td>
<td>816666</td>
<td>816666</td>
</tr>
<tr>
<td>Diff-1</td>
<td>129055508</td>
<td>30266673</td>
<td>11568874</td>
<td>8175276</td>
<td>6822981</td>
<td>10101494</td>
</tr>
<tr>
<td>Diff-2</td>
<td>129055508</td>
<td>60321904</td>
<td>2803742</td>
<td>2844984</td>
<td>4614953</td>
<td>4614953</td>
</tr>
<tr>
<td>TTM</td>
<td>12677916</td>
<td>6093536</td>
<td>1918809</td>
<td>532596</td>
<td>529080</td>
<td>543875</td>
</tr>
<tr>
<td>XTM</td>
<td>12677926</td>
<td>6159988</td>
<td>1931450</td>
<td>538568</td>
<td>529658</td>
<td>575220</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>p (number of processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>206588</td>
</tr>
<tr>
<td>Stat</td>
<td>206588</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>206588</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>206606</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>206606</td>
</tr>
<tr>
<td>RR-1</td>
<td>206588</td>
</tr>
<tr>
<td>RR-2</td>
<td>206588</td>
</tr>
<tr>
<td>Diff-1</td>
<td>211181</td>
</tr>
<tr>
<td>Diff-2</td>
<td>211181</td>
</tr>
<tr>
<td>TTM</td>
<td>206588</td>
</tr>
<tr>
<td>XTM</td>
<td>206588</td>
</tr>
</tbody>
</table>

Table C.44: MATMUL(16) (fine, cached) – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>p (number of processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>1612140</td>
</tr>
<tr>
<td>Stat</td>
<td>1612140</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>1612140</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>1612158</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>1612158</td>
</tr>
<tr>
<td>RR-1</td>
<td>1612140</td>
</tr>
<tr>
<td>RR-2</td>
<td>1612140</td>
</tr>
<tr>
<td>Diff-1</td>
<td>1614775</td>
</tr>
<tr>
<td>Diff-2</td>
<td>1614775</td>
</tr>
<tr>
<td>TTM</td>
<td>1612140</td>
</tr>
<tr>
<td>XTM</td>
<td>1612140</td>
</tr>
</tbody>
</table>

Table C.45: MATMUL(32) (fine, cached) – Running Times (cycles).
<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>12739148</td>
<td>3841845</td>
<td>1129468</td>
<td>408020</td>
<td>215175</td>
<td>188504</td>
</tr>
<tr>
<td>Stat</td>
<td>12739148</td>
<td>3485580</td>
<td>947240</td>
<td>283918</td>
<td>112888</td>
<td>83668</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>12739148</td>
<td>3673909</td>
<td>1532939</td>
<td>437126</td>
<td>171883</td>
<td>125121</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>12739166</td>
<td>3657353</td>
<td>1136724</td>
<td>369822</td>
<td>170071</td>
<td>136810</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>12739166</td>
<td>3657230</td>
<td>1275774</td>
<td>388444</td>
<td>185461</td>
<td>143820</td>
</tr>
<tr>
<td>RR-1</td>
<td>12739148</td>
<td>3763194</td>
<td>1131171</td>
<td>389605</td>
<td>193748</td>
<td>232248</td>
</tr>
<tr>
<td>RR-2</td>
<td>12739148</td>
<td>3763194</td>
<td>1130461</td>
<td>403750</td>
<td>233886</td>
<td>263437</td>
</tr>
<tr>
<td>Diff-1</td>
<td>12968121</td>
<td>7534645</td>
<td>2246912</td>
<td>1101161</td>
<td>515466</td>
<td>577223</td>
</tr>
<tr>
<td>Diff-2</td>
<td>12968121</td>
<td>4250445</td>
<td>1498754</td>
<td>566448</td>
<td>278983</td>
<td>338331</td>
</tr>
<tr>
<td>TTM</td>
<td>12739148</td>
<td>3810409</td>
<td>1157229</td>
<td>412023</td>
<td>209896</td>
<td>192461</td>
</tr>
<tr>
<td>XTM</td>
<td>12739148</td>
<td>3810157</td>
<td>1144804</td>
<td>406853</td>
<td>214161</td>
<td>211593</td>
</tr>
</tbody>
</table>

Table C.46: MATMUL(64) (fine, cached) – Running Times (cycles).
### C.6.2 Variable $t_n$

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>12739148</td>
<td>4307076</td>
<td>1461846</td>
<td>569716</td>
<td>344047</td>
<td>325631</td>
</tr>
<tr>
<td>Stat</td>
<td>12739148</td>
<td>3693069</td>
<td>1042397</td>
<td>334749</td>
<td>149421</td>
<td>124975</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>12739148</td>
<td>4033242</td>
<td>1802870</td>
<td>599083</td>
<td>251095</td>
<td>195476</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>12739166</td>
<td>4033922</td>
<td>1399579</td>
<td>513653</td>
<td>255174</td>
<td>236947</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>12739166</td>
<td>4033922</td>
<td>1399579</td>
<td>513653</td>
<td>255174</td>
<td>236947</td>
</tr>
<tr>
<td>RR-1</td>
<td>12739148</td>
<td>4291014</td>
<td>1376186</td>
<td>499651</td>
<td>284547</td>
<td>385371</td>
</tr>
<tr>
<td>RR-2</td>
<td>12739148</td>
<td>4189308</td>
<td>1417673</td>
<td>582338</td>
<td>399629</td>
<td>434751</td>
</tr>
<tr>
<td>Diff-1</td>
<td>12968121</td>
<td>8455841</td>
<td>2595856</td>
<td>1478620</td>
<td>814831</td>
<td>911810</td>
</tr>
<tr>
<td>Diff-2</td>
<td>12968121</td>
<td>5152450</td>
<td>1959171</td>
<td>819888</td>
<td>488674</td>
<td>434751</td>
</tr>
<tr>
<td>TTM</td>
<td>12739148</td>
<td>4243316</td>
<td>1469110</td>
<td>613282</td>
<td>324196</td>
<td>396301</td>
</tr>
<tr>
<td>XTM</td>
<td>12739148</td>
<td>4224525</td>
<td>1710448</td>
<td>597103</td>
<td>334786</td>
<td>396301</td>
</tr>
</tbody>
</table>

Table C.47: $\text{MATMUL}(64)$ (fine, cached) – $t_n = 2$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>12739174</td>
<td>5479655</td>
<td>1939592</td>
<td>919830</td>
<td>595148</td>
<td>582113</td>
</tr>
<tr>
<td>Stat</td>
<td>12739174</td>
<td>4108928</td>
<td>1296619</td>
<td>436145</td>
<td>226257</td>
<td>207695</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>12739184</td>
<td>4751970</td>
<td>2523022</td>
<td>822453</td>
<td>432767</td>
<td>345164</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>12739202</td>
<td>4694431</td>
<td>1875179</td>
<td>737853</td>
<td>407837</td>
<td>367605</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>12739202</td>
<td>5107929</td>
<td>1899364</td>
<td>824117</td>
<td>474256</td>
<td>460087</td>
</tr>
<tr>
<td>RR-1</td>
<td>12739174</td>
<td>5094939</td>
<td>1763291</td>
<td>738044</td>
<td>461916</td>
<td>679977</td>
</tr>
<tr>
<td>RR-2</td>
<td>12739174</td>
<td>5105081</td>
<td>1887344</td>
<td>912455</td>
<td>716178</td>
<td>1034937</td>
</tr>
<tr>
<td>Diff-1</td>
<td>12968147</td>
<td>10456151</td>
<td>3949960</td>
<td>1964292</td>
<td>1183699</td>
<td>1711399</td>
</tr>
<tr>
<td>Diff-2</td>
<td>12968147</td>
<td>7233875</td>
<td>2945461</td>
<td>1499787</td>
<td>899749</td>
<td>833174</td>
</tr>
<tr>
<td>TTM</td>
<td>12739266</td>
<td>5145965</td>
<td>2007646</td>
<td>938104</td>
<td>539368</td>
<td>502553</td>
</tr>
<tr>
<td>XTM</td>
<td>12739276</td>
<td>5034013</td>
<td>1936568</td>
<td>904656</td>
<td>571756</td>
<td>594734</td>
</tr>
</tbody>
</table>

Table C.48: $\text{MATMUL}(64)$ (fine, cached) – $t_n = 4$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>12739226</td>
<td>7248817</td>
<td>2822331</td>
<td>1573203</td>
<td>1097566</td>
<td>1124149</td>
</tr>
<tr>
<td>Stat</td>
<td>12739226</td>
<td>4940997</td>
<td>1502199</td>
<td>638631</td>
<td>368665</td>
<td>373035</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>12739220</td>
<td>11276943</td>
<td>3475055</td>
<td>844856</td>
<td>492091</td>
<td>679288</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>12739228</td>
<td>6076809</td>
<td>2748770</td>
<td>1184658</td>
<td>719888</td>
<td>674017</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>12739226</td>
<td>6731273</td>
<td>2611055</td>
<td>1413270</td>
<td>907555</td>
<td>957888</td>
</tr>
<tr>
<td>RR-1</td>
<td>12739226</td>
<td>6732020</td>
<td>2541415</td>
<td>1089971</td>
<td>805437</td>
<td>1234849</td>
</tr>
<tr>
<td>RR-2</td>
<td>12739226</td>
<td>6697210</td>
<td>3046785</td>
<td>1529806</td>
<td>1432662</td>
<td>2474518</td>
</tr>
<tr>
<td>Diff-1</td>
<td>12968103</td>
<td>15091855</td>
<td>6980501</td>
<td>4039756</td>
<td>2802365</td>
<td>3605615</td>
</tr>
<tr>
<td>Diff-2</td>
<td>12968103</td>
<td>10664163</td>
<td>5885315</td>
<td>3157172</td>
<td>2189743</td>
<td>2096786</td>
</tr>
<tr>
<td>TTM</td>
<td>12739356</td>
<td>6951140</td>
<td>2942752</td>
<td>1491604</td>
<td>953880</td>
<td>965724</td>
</tr>
<tr>
<td>XTM</td>
<td>12739366</td>
<td>6876345</td>
<td>3394139</td>
<td>1520421</td>
<td>980527</td>
<td>1002713</td>
</tr>
</tbody>
</table>

Table C.49: $\text{MATMUL}(64)$ (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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>Free-Ideal</th>
<th>Stat</th>
<th>P-Ideal</th>
<th>C-Ideal-1</th>
<th>C-Ideal-2</th>
<th>RR-1</th>
<th>RR-2</th>
<th>Diff-1</th>
<th>Diff-2</th>
<th>TTM</th>
<th>XTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$ (number of processors)</td>
<td>1</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>352508</td>
<td>240678</td>
<td>79771</td>
<td>34179</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>187642</td>
<td>59880</td>
<td>27422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>468545</td>
<td>157456</td>
<td>60753</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>256062</td>
<td>136715</td>
<td>60244</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.50: MATMUL(16) (coarse, uncached) – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>Free-Ideal</th>
<th>Stat</th>
<th>P-Ideal</th>
<th>C-Ideal-1</th>
<th>C-Ideal-2</th>
<th>RR-1</th>
<th>RR-2</th>
<th>Diff-1</th>
<th>Diff-2</th>
<th>TTM</th>
<th>XTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$ (number of processors)</td>
<td>1</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2719340</td>
<td>1854235</td>
<td>545222</td>
<td>177033</td>
<td>84106</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1265849</td>
<td>416058</td>
<td>136558</td>
<td>64582</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>3185236</td>
<td>1111024</td>
<td>374400</td>
<td>165215</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>3518236</td>
<td>1111024</td>
<td>374400</td>
<td>165215</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.51: MATMUL(32) (coarse, uncached) – Running Times (cycles).
<table>
<thead>
<tr>
<th>Mgr.</th>
<th>p (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>21361484 14616058 4098472 1252586 465082 251516</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stat</td>
<td>21361484 9901976 3189039 966169 32456 171770</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P-Ideal</td>
<td>21361484 27300058 8394160 2662464 993452 475717</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>21361502 14842476 7144121 2086614 699671 292729</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>21361502 14842476 7143944 2210214 737067 284211</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR-1</td>
<td>21361484 14842428 4234827 1343622 552189 400868</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>21361484 14842428 4234827 1343622 514305 429484</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>21745016 53968520 17628461 11008495 8519769 42984</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>21745016 16109183 8286869 2925353 1518827 1518827</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>21361484 14842060 4234487 1342960 502354 262824</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>21361484 14837264 4234703 1343355 530603 301307</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.52: **MATMUL(64) (coarse, uncached) – Running Times (cycles).**
C.7.2 Variable $t_n$

<table>
<thead>
<tr>
<th></th>
<th>$p$ (number of processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mgr.</td>
<td>1</td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>21525444</td>
</tr>
<tr>
<td>Stat</td>
<td>21525444</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>21525444</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>21525462</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>21525462</td>
</tr>
<tr>
<td>RR-1</td>
<td>21525444</td>
</tr>
<tr>
<td>RR-2</td>
<td>21525444</td>
</tr>
<tr>
<td>Diff-I</td>
<td>21911988</td>
</tr>
<tr>
<td>Diff-2</td>
<td>21911988</td>
</tr>
<tr>
<td>TTM</td>
<td>21525444</td>
</tr>
<tr>
<td>XTM</td>
<td>21525444</td>
</tr>
</tbody>
</table>

Table C.53: MATMUL(64) (coarse, uncached) – $t_n = 2$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th></th>
<th>$p$ (number of processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mgr.</td>
<td>1</td>
</tr>
<tr>
<td>Free-Ideal</td>
<td>21853390</td>
</tr>
<tr>
<td>Stat</td>
<td>21853390</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>21853400</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>21853418</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>21853418</td>
</tr>
<tr>
<td>RR-1</td>
<td>21853390</td>
</tr>
<tr>
<td>RR-2</td>
<td>21853390</td>
</tr>
<tr>
<td>Diff-I</td>
<td>22245878</td>
</tr>
<tr>
<td>Diff-2</td>
<td>22245878</td>
</tr>
<tr>
<td>TTM</td>
<td>21853482</td>
</tr>
<tr>
<td>XTM</td>
<td>21853492</td>
</tr>
</tbody>
</table>

Table C.54: MATMUL(64) (coarse, uncached) – $t_n = 4$ Cycles / Flit-Hop – Running Times (cycles).
<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>22509282</td>
<td>81192208</td>
<td>22817227</td>
<td>7416160</td>
<td>2955165</td>
<td>1834319</td>
</tr>
<tr>
<td>Stat</td>
<td>22509282</td>
<td>41709687</td>
<td>15919325</td>
<td>5205121</td>
<td>1874563</td>
<td>1102101</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>22509276</td>
<td>247271123</td>
<td>91569791</td>
<td>61800665</td>
<td>25048328</td>
<td>12866223</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>22509294</td>
<td>81194714</td>
<td>24282197</td>
<td>8220647</td>
<td>4417729</td>
<td>2019579</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>22509294</td>
<td>81194623</td>
<td>24282131</td>
<td>8220328</td>
<td>3300310</td>
<td>1993246</td>
</tr>
<tr>
<td>RR-1</td>
<td>22509282</td>
<td>81189238</td>
<td>24283122</td>
<td>8222296</td>
<td>3324033</td>
<td>3054916</td>
</tr>
<tr>
<td>RR-2</td>
<td>22509282</td>
<td>81189238</td>
<td>24283322</td>
<td>8222256</td>
<td>3300310</td>
<td>3039472</td>
</tr>
<tr>
<td>Diff-1</td>
<td>22914102</td>
<td>385614536</td>
<td>157274311</td>
<td>107850359</td>
<td>62692673</td>
<td>-- --</td>
</tr>
<tr>
<td>Diff-2</td>
<td>22914102</td>
<td>126621352</td>
<td>71501656</td>
<td>36834573</td>
<td>15225247</td>
<td>-- --</td>
</tr>
<tr>
<td>TTM</td>
<td>22509412</td>
<td>81187665</td>
<td>24281540</td>
<td>8219678</td>
<td>3299472</td>
<td>1998514</td>
</tr>
<tr>
<td>XTM</td>
<td>22509422</td>
<td>81195257</td>
<td>24281956</td>
<td>8220220</td>
<td>3300050</td>
<td>2055491</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>( p ) (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>370684</td>
<td>250140</td>
<td>87530</td>
<td>41128</td>
<td></td>
</tr>
<tr>
<td>Stat</td>
<td>370684</td>
<td>177405</td>
<td>65854</td>
<td>33010</td>
<td></td>
</tr>
<tr>
<td>P-Ideal</td>
<td>370684</td>
<td>269771</td>
<td>122941</td>
<td>43855</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>370702</td>
<td>270194</td>
<td>93391</td>
<td>46055</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>370702</td>
<td>270194</td>
<td>93190</td>
<td>46376</td>
<td></td>
</tr>
<tr>
<td>RR-1</td>
<td>370684</td>
<td>235411</td>
<td>92211</td>
<td>47934</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>370684</td>
<td>235411</td>
<td>91800</td>
<td>51745</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>377815</td>
<td>442983</td>
<td>194642</td>
<td>129980</td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>377815</td>
<td>351660</td>
<td>121258</td>
<td>68191</td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>370684</td>
<td>307706</td>
<td>93874</td>
<td>54221</td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>370684</td>
<td>256891</td>
<td>94969</td>
<td>50456</td>
<td></td>
</tr>
</tbody>
</table>

Table C.56: MATMUL(16) (fine, uncached) – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>( p ) (number of processors)</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>2792044</td>
<td>1890956</td>
<td>579679</td>
<td>192066</td>
<td>95283</td>
<td></td>
</tr>
<tr>
<td>Stat</td>
<td>2792044</td>
<td>1303664</td>
<td>437048</td>
<td>149906</td>
<td>78204</td>
<td></td>
</tr>
<tr>
<td>P-Ideal</td>
<td>2792044</td>
<td>1965900</td>
<td>837430</td>
<td>211729</td>
<td>101752</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>2792062</td>
<td>1967841</td>
<td>591216</td>
<td>197795</td>
<td>102040</td>
<td></td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>2792062</td>
<td>1967841</td>
<td>582517</td>
<td>200821</td>
<td>99186</td>
<td></td>
</tr>
<tr>
<td>RR-1</td>
<td>2792044</td>
<td>1687823</td>
<td>586390</td>
<td>199798</td>
<td>116810</td>
<td></td>
</tr>
<tr>
<td>RR-2</td>
<td>2792044</td>
<td>1687823</td>
<td>586169</td>
<td>207368</td>
<td>129038</td>
<td></td>
</tr>
<tr>
<td>Diff-1</td>
<td>2842495</td>
<td>3225583</td>
<td>1275404</td>
<td>698435</td>
<td>322556</td>
<td></td>
</tr>
<tr>
<td>Diff-2</td>
<td>2842495</td>
<td>2176414</td>
<td>782205</td>
<td>280134</td>
<td>152494</td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>2792044</td>
<td>1896675</td>
<td>581469</td>
<td>206778</td>
<td>112407</td>
<td></td>
</tr>
<tr>
<td>XTM</td>
<td>2792044</td>
<td>1896498</td>
<td>575366</td>
<td>206040</td>
<td>117651</td>
<td></td>
</tr>
</tbody>
</table>

Table C.57: MATMUL(32) (fine, uncached) – Running Times (cycles).
<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>21652300</td>
<td>14761832</td>
<td>4310075</td>
<td>1258299</td>
<td>454422</td>
<td>245181</td>
</tr>
<tr>
<td>Stat</td>
<td>21652300</td>
<td>10051914</td>
<td>3269131</td>
<td>1012307</td>
<td>360160</td>
<td>207622</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>21652300</td>
<td>15057941</td>
<td>6120650</td>
<td>1385310</td>
<td>510794</td>
<td>248728</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>21652318</td>
<td>15066041</td>
<td>4305572</td>
<td>1279048</td>
<td>457170</td>
<td>263063</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>21652318</td>
<td>15066041</td>
<td>4275520</td>
<td>1302630</td>
<td>456183</td>
<td>255201</td>
</tr>
<tr>
<td>RR-1</td>
<td>21652300</td>
<td>12830193</td>
<td>4266631</td>
<td>1265566</td>
<td>490239</td>
<td>389978</td>
</tr>
<tr>
<td>RR-2</td>
<td>21652300</td>
<td>12830193</td>
<td>4277495</td>
<td>1308848</td>
<td>478205</td>
<td>351479</td>
</tr>
<tr>
<td>Diff-1</td>
<td>22042307</td>
<td>24687501</td>
<td>9370095</td>
<td>4527924</td>
<td>1925214</td>
<td>1490239</td>
</tr>
<tr>
<td>Diff-2</td>
<td>22042307</td>
<td>22131834</td>
<td>5515578</td>
<td>1771289</td>
<td>755502</td>
<td>381221</td>
</tr>
<tr>
<td>TTM</td>
<td>21652300</td>
<td>14771918</td>
<td>4205281</td>
<td>1325285</td>
<td>478531</td>
<td>295849</td>
</tr>
<tr>
<td>XTM</td>
<td>21652300</td>
<td>14648811</td>
<td>4183621</td>
<td>1386063</td>
<td>509640</td>
<td>293876</td>
</tr>
</tbody>
</table>

Table C.58: MATMUL(64) (fine, uncached) – Running Times (cycles).
C.8.2 Variable $t_n$

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>21980100</td>
<td>24690836</td>
<td>6839050</td>
<td>2160913</td>
<td>778212</td>
<td>428396</td>
</tr>
<tr>
<td>Stat</td>
<td>21980100</td>
<td>14681413</td>
<td>2191841</td>
<td>673255</td>
<td>446441</td>
<td>359433</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>21980100</td>
<td>24692890</td>
<td>1046311</td>
<td>2490421</td>
<td>873255</td>
<td>584641</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>21980118</td>
<td>24709519</td>
<td>6995476</td>
<td>2202244</td>
<td>798288</td>
<td>483381</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>21980118</td>
<td>24709494</td>
<td>7590497</td>
<td>2149679</td>
<td>788517</td>
<td>441756</td>
</tr>
<tr>
<td>RR-1</td>
<td>21980100</td>
<td>24709900</td>
<td>7033305</td>
<td>2179351</td>
<td>835849</td>
<td>474295</td>
</tr>
<tr>
<td>RR-2</td>
<td>21980100</td>
<td>24709900</td>
<td>7106609</td>
<td>2242132</td>
<td>846729</td>
<td>482915</td>
</tr>
<tr>
<td>Diff-1</td>
<td>22375593</td>
<td>39955831</td>
<td>17147603</td>
<td>7744693</td>
<td>3928718</td>
<td>278098</td>
</tr>
<tr>
<td>Diff-2</td>
<td>22375593</td>
<td>32066133</td>
<td>10217894</td>
<td>3163537</td>
<td>1291874</td>
<td>735381</td>
</tr>
<tr>
<td>TTM</td>
<td>21980100</td>
<td>20236650</td>
<td>7030368</td>
<td>2216803</td>
<td>827828</td>
<td>485835</td>
</tr>
<tr>
<td>XTM</td>
<td>21980100</td>
<td>20236672</td>
<td>7113994</td>
<td>2227559</td>
<td>856485</td>
<td>482915</td>
</tr>
</tbody>
</table>

Table C.59: MATMUL(64) (fine, uncached) – $t_n = 2$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>22635726</td>
<td>34523875</td>
<td>12826214</td>
<td>3907302</td>
<td>1450599</td>
<td>776006</td>
</tr>
<tr>
<td>Stat</td>
<td>22635726</td>
<td>42458765</td>
<td>16329195</td>
<td>5444747</td>
<td>2060857</td>
<td>1270677</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>22635726</td>
<td>43962980</td>
<td>18526642</td>
<td>5067589</td>
<td>1691956</td>
<td>850316</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>22635754</td>
<td>43964034</td>
<td>12670352</td>
<td>3913197</td>
<td>1434321</td>
<td>820946</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>22635754</td>
<td>43996199</td>
<td>13600879</td>
<td>4023379</td>
<td>1453164</td>
<td>824704</td>
</tr>
<tr>
<td>RR-1</td>
<td>22635726</td>
<td>43964713</td>
<td>12738699</td>
<td>4037218</td>
<td>1656624</td>
<td>824704</td>
</tr>
<tr>
<td>RR-2</td>
<td>22635726</td>
<td>43964713</td>
<td>12803942</td>
<td>4328527</td>
<td>1588434</td>
<td>862345</td>
</tr>
<tr>
<td>Diff-1</td>
<td>23043277</td>
<td>75314543</td>
<td>35094607</td>
<td>12392624</td>
<td>6762955</td>
<td>4680617</td>
</tr>
<tr>
<td>Diff-2</td>
<td>23043277</td>
<td>61019441</td>
<td>22382573</td>
<td>6749695</td>
<td>2731156</td>
<td>1470231</td>
</tr>
<tr>
<td>TTM</td>
<td>22635828</td>
<td>34460805</td>
<td>12803942</td>
<td>4328527</td>
<td>1588434</td>
<td>862345</td>
</tr>
<tr>
<td>XTM</td>
<td>22635828</td>
<td>34460805</td>
<td>12803942</td>
<td>4328527</td>
<td>1588434</td>
<td>862345</td>
</tr>
</tbody>
</table>

Table C.60: MATMUL(64) (fine, uncached) – $t_n = 4$ Cycles / Flit-Hop – Running Times (cycles).

<table>
<thead>
<tr>
<th>Mgr.</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free-Ideal</td>
<td>23946978</td>
<td>83546666</td>
<td>23243558</td>
<td>7312834</td>
<td>2729209</td>
<td>1490544</td>
</tr>
<tr>
<td>Stat</td>
<td>23946978</td>
<td>42458765</td>
<td>16329195</td>
<td>5444747</td>
<td>2060857</td>
<td>1270677</td>
</tr>
<tr>
<td>P-Ideal</td>
<td>23946978</td>
<td>14598273</td>
<td>4909278</td>
<td>8785571</td>
<td>3687700</td>
<td>1955760</td>
</tr>
<tr>
<td>C-Ideal-1</td>
<td>23946990</td>
<td>82568456</td>
<td>25688816</td>
<td>7530983</td>
<td>2688134</td>
<td>1546855</td>
</tr>
<tr>
<td>C-Ideal-2</td>
<td>23946990</td>
<td>63498041</td>
<td>24806013</td>
<td>7494771</td>
<td>2728884</td>
<td>1546171</td>
</tr>
<tr>
<td>RR-1</td>
<td>23946978</td>
<td>62516654</td>
<td>23005915</td>
<td>7560244</td>
<td>2969496</td>
<td>2081732</td>
</tr>
<tr>
<td>RR-2</td>
<td>23946978</td>
<td>82569647</td>
<td>24360516</td>
<td>7659290</td>
<td>3101713</td>
<td>2968417</td>
</tr>
<tr>
<td>Diff-1</td>
<td>24377597</td>
<td>165325301</td>
<td>70003655</td>
<td>32052696</td>
<td>14765763</td>
<td>11118528</td>
</tr>
<tr>
<td>Diff-2</td>
<td>24377597</td>
<td>133560749</td>
<td>43959885</td>
<td>1673886</td>
<td>7476504</td>
<td>3892127</td>
</tr>
<tr>
<td>TFM</td>
<td>23947108</td>
<td>64678466</td>
<td>2348204</td>
<td>8317440</td>
<td>2890945</td>
<td>1582895</td>
</tr>
<tr>
<td>XTM</td>
<td>23947118</td>
<td>64677713</td>
<td>24778887</td>
<td>8562822</td>
<td>3086391</td>
<td>1655296</td>
</tr>
</tbody>
</table>

Table C.61: MATMUL(64) (fine, uncached) – $t_n = 8$ Cycles / Flit-Hop – Running Times (cycles).
Bibliography


