INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

DYNAMIC LOAD BALANCING ISSUES IN THE EARTH RUNTIME SYSTEM

by

Kamala Prasad Kakulavarapu

School of Computer Science
McGill University, Montréal
Québec, Canada
December 1999

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Multithreading is a promising approach to address the problems inherent in multiprocessor systems, such as network and synchronization latencies. Moreover, the benefits of multithreading are not limited to loop-based algorithms but apply also to irregular parallelism. EARTH - Efficient Architecture for Running THreads, is a multithreaded model supporting fine-grain, non-preemptive threads. This model is supported by a C-based runtime system which provides the multithreaded environment for the execution of concurrent programs.

This thesis describes the design and implementation of a set of dynamic load balancing algorithms, and an in-depth study of their behavior with divide-and-conquer, regular, and irregular classes of applications. The results described in this thesis are based on EARTH-SP2, an implementation of the EARTH program execution model on the IBM SP-2, a distributed memory multiprocessor system. The main results of this study are as follows:

- A randomizing load balancer with both sender and receiver components using global load state information provides scalable, robust performance for recursive and irregular applications. Furthermore, a randomizing algorithm performs the best as long as the cost of computing the random number does not dominate the overall time of thread execution.

- Load state information outperforms history information for irregular and recursive applications. However for regular applications, history information is more preferable.

- A purely sender-initiated algorithm is the best choice in two scenarios: barrier-synchronized applications, and very fine-grain applications at low input workloads.

- A simple, work-stealing load balancer is preferable for applications with modest thread granularities, and very low workloads.

Other major contributions include:

- Description of a runtime system for a non-blocking, non-preemptive multi-threaded programming model.
- A detailed analysis of costs associated with EARTH operations, and a comparative study of EARTH performance on three different platforms.
- Proposal of a new classification scheme for multi-threaded systems. This is supplemented by an extensive literature survey.

Résumé

Les systèmes concurrents à fil d'exécution multiple représentent une approche prometteuse dans la résolution des problèmes tels que les réseaux ou les latences dûes à la synchronisation inhérents aux systèmes multi-processeurs. De plus, les bénéfice des systèmes concurrents à fil d'exécution multiple ne sont pas limités aux algorithmes basés sur des boucles mais touchent également le parallélisme irrégulier. EARTH, une architecture efficace pour exécuter des fils d'exécution, est un modèle à fil d'exécution multiple qui supporte des fils d'exécution non-préemptifs et à forte granularité. Ce modèle est structuré autour d'un environnement d'exécution basé sur C qui fournit aux systèmes à fil d'exécution multiple la possibilité d' exécuter un programme concurrent.

Cette thèse décrit la conception et la réalisation d'un ensemble d'algorithme dynamique de répartition de charge, ainsi qu'une étude approfondie de leur comportement sur des classes d'applications régulières, irrégulières ou basées sur la notion de la division pour conquérir. Les résultats décrits ici sont basés sur EARTH-SP2, une réalisation du modèle d'exécution de programme de EARTH sur une machine IBM SP-2, un système multi-processeur à mémoire distribuée. Les principaux résultats de cette thèse sont les suivantes:

- Un répartiteur de charge aléatoire, avec un émetteur et un récepteur utilisant l'information de l'état de charge globale, fournit des performances robustes et évolutives pour des applications récursives et irrégulières. De plus, un algorithme aléatoire est le meilleur pour autant que le coût pour calculer le nombre aléatoire ne domine pas le temps d'exécution d'un fil d'exécution.

- L'information de l'état de charge est meilleure que l'historique pour les applications irrégulières et récursives. Par contre pour les applications régulières, l'historique est préférable.

- Un algorithme initié seulement par l'émetteur est le meilleur choix pour deux

scénarios: les applications synchronisées par barrière et les applications à forte décomposition travaillant sur des entrées peu consommatrice en ressource.

- Un répartiteur de charge "voleur de tâche" est préférable pour les applications avec des fils d'exécution de ganularité moyenne et des faibles charges.

Les autres contributions de cette thèse incluent:

- La description d'un environnement d'exécution pour un modèle de programmation à fil d'exécution multiple, non bloquant et non préemptif.
- Un analyse détaillée des coûts associés aux opérations exécutées sous EARTH et une étude comparative des performances de EARTH sur trois platformes différentes.
- La proposition d'une nouvelle méthode de classification pour les systèmes à fil d'exécution multiple, le tout augmenté d'un vaste survol de l'état de l'art.

Acknowledgements

It was a typical summer afternoon in Montréal. I was going to meet Prof. Guang R. Gao for the first time, two months after he agreed to be my external supervisor. After an hour, I was convinced that he is in a hurry to change the world. Today, I am happy to be part of the changed world. I am grateful to Prof. Gao for giving me the opportunity to work in the EARTH project, and for his constant motivation and support. I thank him for arranging my visit to the CAPSL Lab at the University of Delaware, where I could complete a major portion of this thesis. My discussions with him always helped me understand the issues better, and have elevated the contents of this thesis immeasurably. From my association with him, I have learnt a lot both in research, and in real life.

I am fortunate to have worked with Dr. Olivier Maquelin. Olivier introduced me to multithreading, answered my questions on the EARTH runtime system, and guided me in modeling dynamic load balancer behavior in the EARTH system. This thesis drew immensely from his work on the EARTH runtime system. I sincerely appreciate the interest he showed in my career, and his patience with my learning curve. Throughout this thesis, he has been a tremendous source of inspiration, and I value his advice on research and professional skills. Olivier is very hard to emulate, but I will try.

I sincerely thank Prof. Laurie J. Hendren for the financial support despite knowing me only as a former ACAPS lab member. Laurie's kind consideration has helped me a lot during the crucial last semester of my thesis.

Dr. Ruppa K. Thulasiram has been my mentor during my stay at the CAPSL lab. He was always patient to listen to my wild ideas, and offer comments which significantly affected the quality of this thesis. His remarks on the time requirements of this thesis, and his motivation have helped me work hard to complete my thesis in time.

It has been my pleasure to know personally Dr. Kevin B. Theobald. I learnt from his experience on the EARTH project, and his remarks about thesis preparation. I thank him for providing me access to the CAPSL computer systems, and for many quick hacks

which made life before deadlines a little less unpleasant. Outside office, he was lots fun to hang around with, and I will miss the dinner routine.

Dr. José Nelson Amaral made a big impact on my technical writing skills. I have learnt from him better ways to organize ideas, and to present issues in a clear, coherent manner, though I admit I still have a long way to go. I have enjoyed our collaboration on different papers based on contents of this thesis.

This thesis would not have been possible if not for the efforts of former and current members of the ACAPS lab, McGill University, and the CAPSL Lab, University of Delaware. Dr. Olivier Maquelin has implemented the Threaded-C preprocessor and the runtime system. The benchmarks used in this thesis to analyze overheads and latencies of multithreaded operations are also part of his work. I thank Dr. Xinan Tang for pointing us the paper on the supermarket model, which convinced us to go ahead with the randomizing balancer in this thesis. Andres Marquez did an excellent job of presenting our paper on dynamic load balancing at a workshop in Orlando on our behalf. My discussions with Parimala Thulasiraman on Threaded-C programming, and distributed algorithms were very educative. Christopher Morrone is another runtime system person in the group, and we had very interesting discussions about implementation issues of the runtime system. I thank Kevin Theobald, and Chris Morrone for performing some experiments on the MANNA and the Beowulf systems which are used for comparative performance in this thesis. Chrislain Razafimahefa translated the abstract into French in a very short time.

While performing my thesis research at two Universities enriched my experience, it also brought with it quite a few personal down-times. I am fortunate to have friends who helped me in these situations with moral, financial, and logistical support. Parimala and Thulasi were always there to help me. They were instrumental in my maintaining my sanity during many a trying times. Vijay Sundaresan is a great buddy, and I appreciated another cricket enthusiast for company. Besides cricket, Vijay was someone to whom I could always turn to, and he always surprised me with his quick and wholesome support. I enjoyed our long conversations, and heated debates about everything from academics to cricket teams. I had a great time hanging around with Tripat Gill. Tripat is an artist at drinking coffee, and no doubt we spent most of our time in coffee shops. I appreciate his support in more ways than I can count. Chrislain Razafimahefa provided the soccer connection. I could appreciate the significance of World Cup Soccer to this planet after watching him at work on an unrepentant television. He is a very understanding and generous friend. Mike Soss and Tallman were great office mates, and completed the buffet

gatherings. Kunal and Rashmi Gupta are great friends, and I can never forget their assistance in my initial days in North America. Danielle Azar is my officemate and helped me in many ways a friend and an officemate could do. Next time, I should remember to leave the key on her desk. Charles Abety helped me during thesis submission time, and it made a crucial difference. I appreciate the friendship and support of Krishna Mohan, Hari, Suresh, Balaji, and Srinivas; talking to them always made me feel good.

Special thanks are due to Sean Ryan, and Danielle Azar. Sean reviewed the technical memos that were part of this thesis and gave valuable comments that improved their presentation. Danielle allocated more than a fair share of her busy schedule in reading through the final thesis, and helped improve the quality of the presented thesis.

I acknowledge with gratitude the Cornell Theory Center, Cornell University for allowing us access to their IBM SP-2 system, on which the results in this thesis are based upon. I thank the Argonne High-Performance Computing Research Facility, and CACR, Caltech for allowing us access to their IBM SP-2 systems which helped us perform wide-ranging experiments.

I could never hope to reach this stage without the support of Franca Cianci. Franca was very understanding, supportive, and super fast in her replies to my not so uncomplicated questions. I am grateful for the support of Lise Minogue who made signing TA contracts such a pleasure. I thank Lucy St-James for her patience and prompt addressal of my administration related queries. I thank Marilyn Gombe for her help during my thesis submission. I thank Vicki Keirl for providing me the facilities to complete my Masters program.

Finally, I can never repay the debt to my family. They were enormously patient, solidly supportive, and never flinching in their confidence in me. I am immensely grateful to my Parents for their love, support, and encouragement. They always gave me the independence to pursue my choice, but also worked very hard to give me the strength to face the world. In the past few months, my brother Sudhakar turned out to be a great motivator. I can never thank them enough.

To my *Parents*, for their boundless love, support and encouragement

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Designing multiprocessor systems that deliver a reasonable price-performance ratio using off-the-shelf processor [111] and compiler technologies is a major challenge. While modern processors can issue multiple instructions per cycle, they lack the features required to address fundamental issues in multiprocessing systems: latency, bandwidth and synchronization overheads. A well designed parallel system must balance the trade-off between a fine task granularity [142] and the impact of communication latencies on performance. Coarse-grain parallel systems can tolerate long latencies if the application provides enough parallelism because each task is long enough to amortize the communication overheads. But coarse grain systems do not fully exploit the parallelism in irregular applications. Fine-grain parallelism, on the other hand, enables further parallelization of many applications, but has proved to be difficult to support due to the higher relative cost of communication latencies [142].

Multi-threading is a promising approach to overcome two major pitfalls of conventional parallel computing, and in particular fine-grain parallelism - communication and synchronization latencies [30, 13, 10, 49, 55, 131, 75, 83, 82, 170, 112, 72, 76, 102, 106, 134, 138, 148, 149, 137, 162, 161]. Multi-threaded languages efficiently manage the low computation to communication ratio (R/C) in fine-grain parallelism by supporting several threads of control per node and switching to a new thread whenever a long latency operation is encountered. The fine-grain threads offer better expressiveness, low thread management overheads, and higher processor utilization. These features facilitate

significant performance improvements for all classes of applications, including the irregular and dynamic applications which are difficult to program efficiently with coarse-grain parallelism.

Messina *et al.* [115] study the current trends in high performance computing, and evaluate future requirements in architecture and programming models in order to sustain current tempo in system performance into the next decade. According to this study, Moore's law will hold true for at least two more generations, until feature sizes of 0.08-0.13 μm are reached. At this point in 2010, the limits of CMOS silicon will have been reached[1]. One of the approaches suggested, besides developing alternate component technologies, is multithreading. Multi-threading allows hiding of the rapidly increasing disparity between processor and memory speeds. Work is decomposed into individual tasks (threads), which are scheduled on processors after their data/synchronization constraints are met. One requisite condition for hiding memory latencies is that the ratio of threads to processors should be high enough so that there is always work to be done by a sufficient number of ready threads.

Recent studies have shown that it is possible to support fine-grain multi-threading efficiently with off-the-shelf technology [112, 156]. According to this study [153], three features are required for efficient runtime support of fine-grain threads. Threads should be *abundant*, *balanced*, and *cheap*. Having an abundant number of active threads on a processor increases processor utilization, because if one thread is delayed, another thread can start execution. A large pool of threads also offers good potential for load balancing. Economic load balancing is essential in order to adapt to dynamic application behavior at runtime. Finally, thread creation, termination, synchronization, and context-switching should be cheap.

Using off-the-shelf technology implies ruling out custom hardware in building multi-threaded systems. An alternative is to emulate the multi-threaded model in software. A runtime system for multithreaded systems assumes the responsibility to provide an ideal interface between the multithreaded code and the hardware platform, and implements an environment for efficient execution of threads. A significant component of the runtime system is the dynamic load balancer. The dynamic load balancer reacts to load imbalances at runtime, and aims to keep all the nodes busy. Past studies in dynamic load balancing have focussed on two objectives: keeping all the nodes busy; and optimizing

[1]According to the Semiconductor Industry Association, the clock speed for high performance microprocessors between now and 2010, is expected to increase from 500 MHz to 1,100 MHz [115].

load balancing by minimizing balancer overheads and maximizing benefits due to load balancing. However, the understanding gained so far has been limited to distributed computing, and these results are to be studied in the context of multithreaded systems, where dynamic task generation, fine-grain parallelism impose challenging constraints on the dynamic load balancer. Furthermore, maximum utilization of CPU cycles for application workload becomes even more hard to achieve for distributed memory based applications whose behavior is difficult to predict at compile-time.

Section 1.1.1 introduces parallel job scheduling in multiuser environments. Various issues involved in building a dynamic load balancer are studied in section 1.1.2. Section 1.1.3 introduces the current status of the EARTH project, and the role of dynamic load balancing in the EARTH program execution model. Section 1.2 lays down the case for a better understanding of load balancer behavior in fine-grain multithreaded systems, and the aim of this thesis is stated in section 1.3. Section 1.4 concludes the chapter by summarizing the contributions of this thesis.

1.1.1 Parallel Job Scheduling

Scheduling in uniprocessor systems decides the next thread which is to be allocated CPU time. In multiprocessor systems, additional aspects of scheduling have to be considered: where, when, and which thread. These decisions can be made by the operating system, by the language runtime system, or by the application itself. There is an additional scheduling decision to be made in multiuser, multiprocessor systems - resources have to be allocated to the application before starting its execution.

Scheduling in multiuser environments usually is the combination of two actions: first, allocating resources for an application execution, and second, deciding the next thread to execute in a pool of ready threads. The second stage is similar to *dispatching* in operating systems. Another runtime aspect of parallel execution that operates at one level higher than dispatching is dynamic load balancing. The goal of dynamic load balancing is to ensure maximum possible utilization of the CPU resources. This is different from the objective of selecting a particular thread for execution. While dynamic load balancing aims to ensure that all processors are busy with adequate workloads on each node, scheduling selects the next available thread to execute.

Two approaches to scheduling jobs[2] in multiuser multiprocessor systems are studied [54]. With *single-level* scheduling, the operating system performs the actions of allocating resources for an application, and allocating CPU time to competing threads in the same job. These two actions are decoupled in *two-level* scheduling, where, the operating system allots the resources to a job, and the scheduling of threads is done by a higher-level software, either by a runtime system or by the application itself. A major implementation related distinction between the two scheduling policies is partitioning of resources along the time or space axis. Space-slicing requires exclusive allocation of resources, leaving the operating system with less control. Time-slicing, on the other hand, is more flexible, but comes with higher overheads.

Single-level scheduling causes operating system overhead for every scheduling decision, and this is very costly for fine-grain applications with a high number of synchronizations. On the other hand, two-level scheduling is more suited for shared memory machines rather than for distributed memory architectures, especially if the programs are written in the SPMD style.

A popular approach is to partition the resources among jobs, and then run a single thread on each processing element. This is easy to implement, and suits SPMD programs which run in batch mode. This approach allows dedicated access to multiprocessor machines for parallel applications. The operating system sees only a single thread, which is the user-level runtime system. Application threads are invisible to the operating system, and their management, synchronization, and load balancing is performed at application level in the user space. Parallel applications in the EARTH system are executed in this manner.

Another important feature to be considered in parallel applications is the type of jobs with respect to the processor allocation [53, 54]. Jobs can be of the following types: *Rigid* jobs require a certain number of processors. They will not run on fewer, and will not utilize more. *Moldable* jobs allow the number of processors to be set at the outset, but it cannot be changed thereafter. *Evolving* jobs have changing requirements, for instance a sequence of serial and parallel phases. At the beginning of each phase, the job requests the system for the resources it needs for this phase, and at the end of the phase it releases them. Jobs submitted for execution in the EARTH system belong to the moldable category. Dynamic load balancing has very high potential in this type of jobs because, application performance depends on the ability of the balancer to map available concurrency in the

[2] A job is an application in execution, as known by the operating system.

application onto varying number of processors efficiently.

Finally, the initial placement of threads determines the job elapsed time, and thereby the importance of dynamic load balancing in the system. Load-balancing is most crucial in systems where all the threads are placed in a single node initially. In the case of EARTH, the first thread is placed on node 0, and this thread generates the parallel workload, which is subsequently distributed across all the nodes in the parallel execution by the dynamic load balancer. Applications which have a sequential-parallel-sequential phases of computation represent another domain of applications where load balancing assumes an important role. For instance, in a barrier-synchronized application, node 0 generates parallel workload, and waits till all the nodes complete, and then issues the next set of work. In this case, very fast load distribution is required.

1.1.2 Dynamic Load Balancing

The total elapsed time of an application running in parallel over multiple nodes is limited by the slowest node. One way of speeding up the program execution is to allow equal distribution of workloads on all the processors, so that the prospect of few nodes executing most of the parallel workload while other nodes are idle is avoided. *Load balancing* is the strategy used to minimize the total execution time, by distributing workload equally over all the nodes participating in the execution.

In distributed systems, an attempt to distribute workload equally involves very high computational overheads. Most of the work is spent on collecting global state. If the applications considered demonstrates a pattern of frequent communication and synchronization, this global state changes rapidly, making load balancing unviable. Furthermore, if the grain size of the transfered work is not big enough to amortize the load balancing overheads, load balancing is not preferable even if the balancer algorithm guarantees accurate decisions based on global system state. In these situations, an alternative to distributing workload equally, is to ensure that all nodes are *busy*. Reducing idle times, and thereby the total program execution time is a far more preferable objective than attempting to distribute the workload equally. This strategy is called *load sharing* [136]. Most systems including the EARTH system implement load sharing rather than load balancing. These two terms are now being used interchangeably. The term load balancing should be read as load sharing in the EARTH system.

Load balancing can be performed at compile time and runtime. Static load balancing is done by the programmer/compiler, and is more suitable for regular applications where it is possible to predict communication patterns, and also where moving workload at runtime entails a huge cost due to data locality. However, static load balancing is not suitable for dynamic, irregular applications, where it is not possible to predict not only the communication patterns of applications, but also the grain size of the workload on each node. Dynamic load balancing algorithms use system state information in making runtime decisions in migrating workloads. An unavoidable consequence of this reliance on dynamic system state is the high overheads associated with this approach. In addition, care should be taken to avoid using old state information in order to avoid potential inaccuracies in balancer decisions, and as a result instability in the system. It is a challenging task therefore, to design dynamic load balancer algorithms that take into account system state, application behavior, and make quality decisions at minimum overheads to ensure minimum idle times.

In an ideal scenario, adaptive load balancing algorithms provide the best possible performance. These algorithms adapt to the global state by changing their policies, and algorithms at runtime. It is well known that it is difficult to have a good load balancer for all applications. With adaptive algorithms, it is possible to switch to the appropriate balancer at runtime, as a response to change in application load conditions. However, these balancers are difficult to implement.

A typical load balancer algorithm has four phases - processor load evaluation, load balancing profitability determination, task selection, and task migration [168, 165]. For systems where load has to be distributed equally, the task selection phase has in turn two stages: in the first stage, the amount of workload to transfer in order to achieve system-wide load balance, is computed; in the second stage the actual tasks whose computation time represents the difference of the workloads, are selected.

The different phases of the load balancer are implemented by the four components in any load balancer [136]:

Transfer Policy : A transfer policy determines whether current load state on a node warrants the initiation of task transfer, with the node either as a sender or as a receiver. Usually the transfer policy is based on *threshold* policy. The state of a node with respect to load balancing is determined as per the values of predetermined upper and lower thresholds.

Selection Policy : This policy identifies a task for migration. Several factors are considered in task selection. Firstly, the overhead due to task transfer should be minimal. Secondly, the task should execute long enough to amortize the transfer overheads. Finally, any location dependencies should be maintained.

Location Policy : A partner node for the task migration is identified with the location policy.

Information Policy : The information policy decides *when* information about other nodes is to be collected, from *where* it is to be collected, and *what* information is to be collected. Three types of information policies are reported in the literature. A *demand-driven* policy allows a node to collect load information only when it needs to transfer work. With the *periodic* policy, a node collects load information periodically, depending on the information collected, the transfer policy may decide to initiate task transfer. Finally, under the *state-change-driven* policy, nodes dessiminate their load information when their states change by a certain degree. This policy differs from the demand-driven policy in that nodes dessiminate their load information, in contrast to soliciting load information of other nodes.

The transfer policy determines the mode of the balancer, either as a sender or receiver. This brings up the issue of balancer initiation. Balancers can be *receiver-initiated* (work-stealing), or *sender-initiated* (work-sharing), or *hybrid* (symmetric) [136, 33]. Receiver-initiated load balancers transfer the load balancing overheads onto the idle node. Because the load balancing actions are triggered by change in local state, this approach results in minimum overheads. On the other hand, sender-initiated balancers dispose their extra workload onto other nodes in the system. This strategy may result in instability of the system, due to multiple redundant load balancing actions on all the nodes in the system. The hybrid balancers seek to include the advantages from both receiver-initiated and sender-initiated balancers, and are usually preferable for all load conditions.

The topology of the interconnection network assumed in the load balancer model also plays an important role in performance. Typical interconnection topologies are mesh, ring, complete graph, and hypercube. Besides their influence on node-to-node communication latencies, routing of the load balancing messages through the interconnection network, opens the possibility of collecting load state information effortlessly.

1.1.3 The EARTH System

EARTH (Efficient Architecture for Running Threads) is a parallel multi-threaded environment developed at McGill University [84, 82, 156, 153], and is now an active research topic at the University of Delaware. The EARTH programming model has been implemented on several existing, conventional multiprocessors such as MANNA (developed at GMD-FIRST, Germany), IBM SP-2, Beowulf, and Sun SMP Cluster. The research areas pursued in EARTH are architecture design, run-time systems, dynamic load balancing, parallelizing compilers, and parallel applications [160, 159, 158, 72, 98, 14, 153]. The program execution model of the EARTH system is a crucial component in the Hybrid Technology Multi-threaded Architecture project (HTMT) [67, 154, 62, 64].

Multi-threaded programming support can be provided in two ways. One possibility is to make the threads explicit using a threaded extension to a general purpose language. This choice gives the programmer more freedom, allows expressiveness and efficiency for multi-threaded programs [151]. An alternative is to provide a more traditional high-level language together with compilation techniques to automatically generate multi-threaded code [75]. EARTH-C is a user-friendly language that is automatically translated to EARTH Threaded-C, an explicitly parallel language. The translation sequence, including code generation for the target machine is summarized in Fig. 2.1.

Initially, research in the EARTH model was based on the MANNA platform. Later, a portable implementation of the EARTH environment was developed in order to support a larger number of target architectures. Currently, significant effort is being directed into providing efficient multi-threading support, even on conventional multiprocessors and networks of workstations.

Dynamic load balancing was part of the EARTH programming model from the beginning. It is tightly integrated into the run-time system, which manages both descriptors for threads that are ready to execute, the *Ready Queue* (RQ), as well as the units of work used for load balancing: the *Tokens* stored in the *Token Queue* (TQ). Fig. 2.18 shows how these two queues are interconnected, as well as the interface to the node's CPU and the interconnection network.

The load balancing policy supported by default by the run-time system has proved to work fairly well on a large number of applications. However, there is still room for improvement [33]. Moreover, as devising a general-purpose and efficient load balancing policy is not an easy task, we think it is necessary to get a better understanding of dynamic load balancing behavior in the context of multi-threaded multiprocessor systems.

1.2 Motivation

Previous studies [33] have shown that it is difficult to come up with one load balancer that suits all applications. While this work has been a very good starting point for gaining an understanding of load balancer behavior for fine-grain multi-threaded systems, much work needs to be done to develop scalable and stable load balancers, and understand balancer behavior for both commonly occurring, and extreme load imbalances. Most importantly, the understanding gained from past work in dynamic load balancing for process-based distributed computing needs to be evaluated in a multi-threaded context.

Our experiments have shown varying performance for the different load balancers. The unpredictable behavior of fine-grain threads in dynamic applications, as well as the multiple dependences among threads running on different nodes make it difficult to come up with a complete analytical model of the load balancer behavior. Nevertheless, several aspects of dynamic load balancing can be empirically studied, allowing us to make predictions on the system behavior.

Benchmark	Dual	Spn	Shis	Snd	His	Range	Catapult	Rand
Fibonacci(33)	3	2	7	8	4	6	5	1
Queens(12)	6	2	7	3	5	4	8	1
TSP(10)	7	5	8	6	3	4	2	1
Knary(7, 7,2)	7	3	8	6	2	4	5	1
Matrix(1024X1024)	7	5	8	4	1	2	6	3
Tomcatv(257)	4	3	8	8	2	1	8	5
SPMD(4,4,0)	6	2	8	1	4	5	7	3
Paraffins(28)	5	3	7	6	2	4	8	1
Average	5.6	3.1	7.6	5.3	2.9	3.8	6.1	2
Rank	6	3	8	5	2	4	7	1

Table 1.1: Relative ranking of the different balancers based on their elapsed times as shown in Table 5.1.

Threaded-C programs written for divide-and-conquer, regular, and irregular classes of applications are executed with eight dynamic load balancers in the EARTH system. Table 5.2 shows individual rankings of the balancers for each of the applications. A ranking of the balancers for this set of applications is provided in the last row. The results for the first seven balancers show that while there is significant improvements over a

"no-load balancing" situation, there is no consistent winner for all applications. Furthermore, balancer performance is not consistent even across applications belonging to the same programming model. Here, a few points are worthy of further discussion: Is the system behavior predictable? How would these balancers perform when the load parameters like input workload, number of nodes, application grain size, programming model of the application, or architectural parameters like polling interval, etc. are varied? Is it possible to achieve better performance? The rankings for the last balancer - *Rand* indicate its ability to achieve better, consistent, and scalable performance. It is possible to deduce after considerable experimentation that the *Rand* balancer performs very well for the divide-and-conquer, and irregular classes of applications, while it is not preferable for regular applications. Similarly, what would be a preferable balancer for very fine-grain applications with minimal amount of exploitable parallelism? Or, what is the preferable balancer for barrier-synchronized applications? Is it possible to implement randomizing algorithms in fine-grain multi-threaded systems? Considering that a typical randomizing function costs around 22 μs, what strategies are required to achieve good speedups?

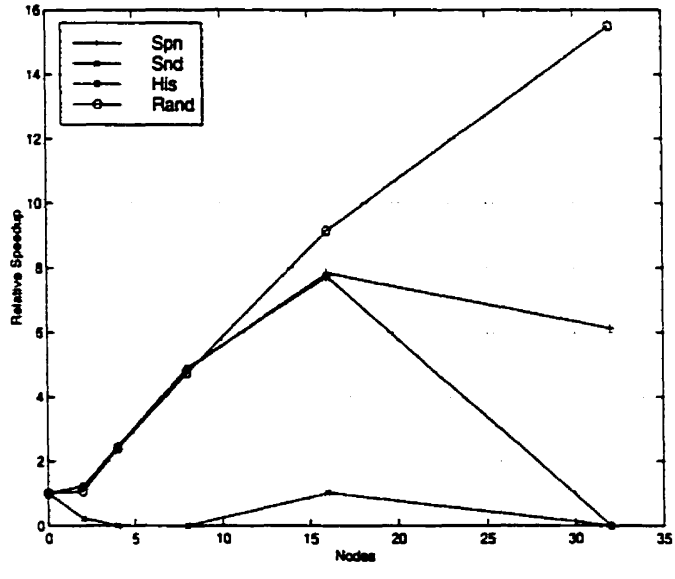


Figure 1.1: Performance of different balancers for Fibonacci(28).

Fig. 1.1 shows the scalability of different balancers for the Fibonacci (28) benchmark. The Threaded-C program for the Fibonacci comprises of very fine-grain threads (approx. 2 μs), and the only work performed in these threads is to spawn parallel workload. The Fibonacci benchmark represents one extreme of fine-grain applications. It is difficult to

achieve scalable performance for applications like Fibonacci, without any programmer effort to coarsen the grain size. However, it is equally important to show that dynamic load balancing is an asset for such fine-grain applications. The *Rand* balancer performs very well relative to the other balancers, specially the *His* balancer[3] both in terms of speedup, and scalability. It is therefore worthwhile to investigate the impact of different balancer policies on widely varying load situations.

This dissertation attempts to answer the above questions. Our aim is to perform a comprehensive study of load balancing for fine-grain multi-threaded systems in terms of algorithms, applications, and architecture. As a part of this study, we have made a comparative study of different load balancer policies, understood their behavior at varying load parameters, and have suggested the suitability of appropriate balancer for diverse load situations. On the basis of these results, we have implemented a dynamic load balancer algorithm that is scalable and robust enough under varying circumstances.

In addition, we describe the implementation of a runtime system for a non-blocking, non-preemptive multi-threading model, and analyze the latencies and overheads of various multi-threaded operations. Finally, we provide the results of an extensive survey into related work in multi-threaded systems, and their dynamic load balancing policies.

In conclusion, our research is leading to a better understanding of dynamic load balancing policies and their impact on application performance. This research should also be applicable to similar parallel systems based on multi-threading and will hopefully allow future systems to achieve better performance on a broad range of algorithms.

1.3 Problem Statement

- *To design and implement a scalable, efficient, consistent, and robust load balancer for fine-grain multi-threaded systems.*

  A runtime system with an efficient balancer shows significant performance improvements over another system with *no* load balancer. Performance should scale well across a wide range of nodes, with minimum degradation in performance as the number of nodes is increased. Consistency of the balancer in all load situations improves predictability of the load balancer behavior. A robust balancer does not cause instability in the system, and always terminates in a deterministic manner.

[3]The *His* balancer is shown as the best balancer for different applications in [33].

- *To study the impact of balancer algorithms for different load situations, and suggest appropriate balancer policy for an application with approximate amounts of parallelism, task grain size, task generation rate, and synchronization patterns.*

1.4 Contributions

We have implemented eight dynamic load balancers, and compare their performance against seven existing balancers. Initially, we study the advantages of different dynamic load balancer policies against a situation where there is no load balancing. We then study the benefits of a randomizing load balancer in a fine-grain multithreading environment with varying application and workload parameters, and compare its performance against seven existing balancers. Next, we look at the different factors that have contributed to the relatively better performance of the randomizing algorithm, by comparing it against different versions of itself, each with varying degrees of sophistication. Finally, we study the influence of various program, architecture, and implementation related parameters on program performance.

The main results of this study are as follows:

1. For irregular and highly recursive programs, it is beneficial to generate large (abundant) number of threads to facilitate the work of the load balancer.
    - Furthermore, a randomizing algorithm (*Rand*) performs the best as long as the cost of computing the random number does not dominate the overall time of thread execution.
    - When this is not favorable for applying the *Rand* balancer, a hybrid history information based algorithm (*His*), a simple work-stealing algorithm (*Spn*) are best suitable in decreasing order.
2. The *Rand* balancer is "good" for fine-grain applications. An in-depth study of the *Rand* balancer performance in different load scenarios is performed
3. When the *Rand* balancer does not perform well, the suitability of alternate balancers is examined.
4. In order to understand the different factors that contribute to the good performance of the *Rand* balancer, a comparative study of the *Rand* balancer with different versions of itself each with varying degrees of sophistication, is performed.

5. Design of a spectrum of experiments to understand application behavior with different load balancers.

Other contributions of this thesis include the following:

1. Description of the runtime system for a non-blocking, non-preemptive multithreaded programming model. Implementation of an elaborate profiling framework in the runtime system, which will aid in better understanding of the time spent in various runtime system activities during program execution.

2. Implementation of a balancer - *Minima*, to provide a lower bound for parallel performance.

3. A detailed analysis of costs associated with EARTH operations.

4. Proposal of a new classification scheme for multi-threaded systems. This is supplemented by an extensive literature survey.

1.5 Thesis Organization

The Threaded-C programming model, and the implementation of the runtime system are described in chapter 2. Chapter 3 introduces the dynamic load balancing algorithms designed and implemented in this thesis. The experimental framework is discussed in chapter 4. Chapter 5 analyzes the performance of different balancers for fine-grain applications. Chapter 6 discusses the costs and overheads of various EARTH operations on the IBM SP-2 system. A comparative study of performance of EARTH implementations on three different distributed memory platforms is studied in chapter 7. Related work, both in in terms of runtime systems for multithreaded models, and different dynamic load balancing techniques, is studied in section 8. Appendix B studies the parallel environment in the EARTH system, and gives easy illustration of typical EARTH operations like invoking local functions, or performing data synchronization. Appendix C lists the profiling support built into the EARTH runtime system. Appendix E lists the results of some additional experiments conducted.

Chapter 2

The EARTH Multithreading System

EARTH - Efficient Architecture for Running THreads [84, 150] is a multi-threaded architecture and execution model that supports fine-grain, non-preemptive threads and allows the implementation of a multi-threaded execution model with off-the-shelf microprocessors in a distributed memory environment. In order to reduce OS related costs, EARTH threads operate at the user-level. The EARTH runtime system assumes the responsibility to provide an interface between an explicitly multi-threaded program and a distributed memory hardware platform. The runtime system performs thread scheduling, context switching between threads, inter-node communication, inter-thread synchronization, global memory management, and dynamic load balancing.

In the EARTH architecture, applications are written in Threaded-C [152], a multi-threaded variant of C. Threaded-C can also be used as a compilation target for other parallel languages [74]. Threaded-C provides constructs for the definition of fine grain, non-preemptive threads, for the specification of data transfers, and for synchronization among threads. In Threaded-C computations may be composed from arbitrary function call graphs. Multiple threads can be enabled simultaneously either because data is produced or because synchronization signals arrive. Alternatively, threads may also be explicitly spawned. Threaded-C implements a global memory space comprising the local memories on all nodes in the system.

The translation sequence for programs written in Threaded-C is shown in Fig. 2.1. Threaded-C programs are first preprocessed into sequential C programs by the Threaded-C preprocessor (`etcpre`). Each of the threads is transformed into a separate C function, with the Threaded-C constructs replaced by equivalent C code according to their semantics. The preprocessed code is compiled to object code with a traditional C compiler. The

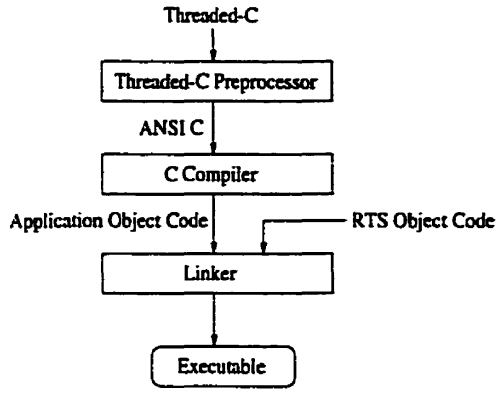


Figure 2.1: Translation Sequence of Threaded-C code

final executable is obtained by linking the application object code with the runtime system object code.

Communication latencies associated with remote operations pose a challenge to implement fine-grain parallelism in a distributed memory platform. Implementing efficient communication on EARTH is important because of its fine-grain threaded model, where the threads can be very short (typically a few hundred μs on the IBM SP-2). The EARTH runtime system seeks to minimize the overheads involved in data communication, synchronization, and load balancing.

This chapter presents a description of the portable EARTH runtime system. The implementation of the runtime system from the Threaded-C language - runtime system boundary to the final execution of code is studied. Initially, we look at the preprocessed code from Threaded-C to see the realization of Threaded-C constructs in the runtime system. This is followed by a detailed explanation of the strategies adopted to implement the runtime system functionality.

The rest of the chapter is organized as follows. Section 2.1.1 describes EARTH programming model and illustrates the Threaded-C language for the Fibonacci example. Section 2.2 describes the translation of Threaded-C constructs into sequential C and the emulation of a global address space on a distributed memory architecture. Section 2.3 describes the implementation of the EARTH runtime system, including its dynamic load balancing algorithms. Scheduling of threads is explained in section 2.3.2. The runtime system behavior while implementing two EARTH operations is traced in section B.

2.0.1 Current Implementations

EARTH is currently implemented on multiple platforms - network of Sun workstations, MANNA, IBM SP-2, Beowulf, and a SUN SMP cluster. All platforms except for the Sun SMP cluster are distributed memory implementations. The core of the runtime system is the same on all platforms. Specific interfaces with the CPU and the network are implemented for each platform. The portable runtime system stresses minimum interactions with the hardware, as might be observed in minimum amount of assembly code and references to machine specifications. Unlike the platforms mentioned above, the runtime system for Manna is not portable. However, portable Threaded-C programs can execute on the Manna version of the runtime system.

Earlier studies on EARTH [84, 111] described the implementation of the EARTH model on the MANNA machine, which has two processors in each processing node. In such a platform one processor can execute threads while the other is in charge of inter-node communication, synchronization and dynamic load balancing. In this dissertation we report results of the EARTH implementation on the IBM SP-2 that has a single processor per processing node. In this implementation all the activities of the EARTH model are supported by the same processor.

2.1 Threaded-C

Threaded-C was originally conceived as an intermediate language for a higher level parallel language - the **EARTH-C** [74] as part of the EARTH project [84]. As the name suggests, Threaded-C extends the C language with threaded constructs. With gradual improvements, Threaded-C emerged as a programming language in its own right. It is *complete* in the sense that, it provides for constructs that fully capture the multithreading model, giving the programmer complete control of thread construction and thread launching.

Some salient features of the Threaded-C language are as follows:

- *Fine grain, non-preemptive, non-blocking* threads perform the multithreaded computation.
- Multithreaded computations may be composed from arbitrary call graphs.

- Rich semantics associated with the Threaded-C constructs support thread definition, parallel invocation, synchronization and inter-thread communication.
- Multiple threads can be launched simultaneously, either on the receipt of relevant data for which the threads are waiting (in a dataflow-like fashion), or by synchronization signals sent to the waiting thread by the programmer. Alternatively, threads may also be explicitly spawned.
- Applications execute in a global memory space comprising the local memories on all nodes in the system. Local and global pointers on Network of Workstations can be accessed in the programs.

Multithreaded computation on NOW provides for temporal and spatial parallelism. The partial ordering of instructions, unlike in sequential programming, allows for considerable dynamic behavior. This dynamic schedule is governed by the RTS at run-time, while the language helps specify a call graph sequence at compile-time.

2.1.1 Programming Model

A *thread* is a set of instructions that is executed sequentially in an atomic fashion. Fine-grain, non-preemptive threads which when started execute till completion, are the atomic units of multithreaded computation in EARTH. Interacting threads sharing context are grouped into bigger units - *threaded functions* [151]. Every thread in a threaded function is numbered, with numbers starting from 0. The execution of a threaded function always starts from thread 0. Further-on, the term 'function' is to be read as threaded function unless otherwise specified.

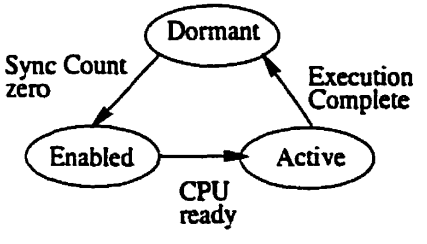


Figure 2.2: Thread States

Threads are enabled for execution through synchronization signals. A threaded function can allocate an array of sync slots. Typically each one of the slots in the array is

associated with a different thread and the slot counter is initialized with an initial value. When the arrival of a signal causes the counter of a slot to reach zero, the runtime system moves the thread associated with the slot from the *dormant* state to the *enabled* state, and resets the counter to a pre-specified reset value.

In Threaded-C, the producer and consumer parts of the code are split into two threads, both of which are linked by synchronization slots. The sync slots mechanism provide a unique handle to address individual threads, enabling the definition of any arbitrary thread activation graph. Fig. 2.2 shows the various states associated with a thread. Initially the thread is in the *dormant* state when it is yet to receive certain number of synchronization signals required for its execution. Synchronization signals may or may not be preceded by any data for the thread. After receiving the required number of synchronization signals, the thread is enabled and placed for execution in a ready queue. When the CPU is available, the thread moves to the *active* state where it starts execution. Once a thread is active, it must run to completion without preemption. The *thread boundary* is a point in a threaded function where one thread finishes, and another starts. At the thread boundary, while the consumer thread is waiting for a sync signal, another *active* thread is executed. The *split-phase* nature of EARTH operations overcomes communication latencies by switching to another *active* (or enabled) thread, while servicing the communication requirements of another thread.

The *context* for a threaded function includes the array of sync slots, the function arguments and the local variables. At any instant of time, only one thread is running on a processor, though there may be multiple threads belonging to the same application running on multiple processors. A detailed explanation of the portable Threaded-C language model is given in [151, 146].

Parallelism is realized through threaded function calls. When a threaded function is invoked, the caller and the callee execute concurrently (if there are available CPUs). Execution of a multi-threaded program with concurrent function invocations leads to dynamic unfolding of the computation and the activations form a tree referred to as *activation tree*. The nodes and edges of this tree represent the threaded functions and their synchronization dependences respectively. After a function invocation, both the caller and the callee may run in parallel. All the function frames are active, and the caller continues execution after invoking the callee. This is in contrast to the normal sequential function call mechanism, where the caller suspends until the callee returns.

A sample threaded function that computes the n^{th} element of a Fibonacci sequence is

```
THREADED fib (SPTR done, int n, int *GLOBAL result)
{
 SLOT SYNC_SLOTS [1];
 int r1, r2;

 INIT_SYNC(0, 2, 2, 1);
 if (n < 2) {
     r1 = 1; r2 = 0;
     SPAWN(1);
 } else {
    INVOKE(0, fib, SLOT_ADR (0), n - 1, TO_GLOBAL (&r1));
    TOKEN(fib, SLOT_ADR (0), n - 2, TO_GLOBAL (&r2));
} END_THREAD();

THREAD_1:
 DATA_RSYNC_L (r1 + r2, result, done);
 END_FUNCTION ();
}
```

Figure 2.3: Parallel Function Invocation in Fibonacci Program

shown in Fig. 2.3. The THREADED keyword indicates that the function `fib` is a threaded function, and its activation frame is allocated on the heap. The SPTR keyword is a type definition for a synchronization slot. After declaring an array of sync slots (in this case one slot), the sync slots are associated with threads by the `INIT_SYNC(slot_num, cnt, rst, th_no)` statement, where `cnt` is the initial count and `rst` is the reset count. In Fig. 2.3, sync slot 0 is associated with thread 1. Thread 1 will be ready for execution after receiving two synchronization signals. The SPAWN primitive moves a local thread from the dormant to the enabled state. The parent thread continues execution after executing SPAWN. The INVOKE and TOKEN constructs are used to launch child threaded functions that run in parallel with the parent threaded function. When the TOKEN construct is used, the processing node that will execute the function is decided by the dynamic load balancer at runtime, whereas the INVOKE construct specifies the processing node that must execute the function as its first argument[1]. The DATA_RSYNC_L primitive places the data (`r1 + r2`) at the destination memory location pointed to by the global variable

[1]Normally, the first recursive call in Fig. 2.3 would also be issued with the TOKEN construct to ensure maximum dispersability of the tokens. In this example we are using the INVOKE construct to illustrate its use. In order to show all parallel constructs, an optimized version is not presented here.

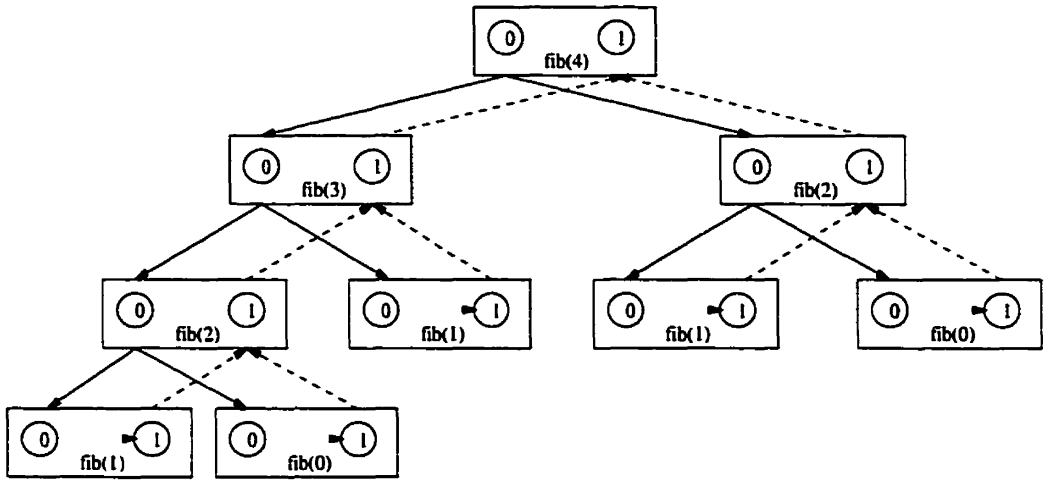


Figure 2.4: Activation Tree for Fib(4)

`result` and sends a synchronization signal to the sync slot pointed to by the variable `done`. The last statement to be executed in the threaded function is the `END_FUNCTION` construct, which results in the deallocation of the activation frame.

Threads are in the dormant state when expecting results from a subcomputation or during communication latencies while waiting for data. Any arbitrary call sequence may be launched by manipulating the *sync slots* (or the synchronization slots). Any synchronization, whether related to computation or communication, is performed through the sync slots. The declaration for the sync slots is expected to be the first statement in a threaded function with more than one thread. All threads that are to be enabled with the sync edges, except for thread 0 which is by default the *starting thread* in a threaded function, are associated with a sync slot by the statement `INIT_SYNC(slot_num, cnt, rst, th_no)`. Another exception to this rule are the threads that may be explicitly spawned, using the `SPAWN` construct. The value `cnt` indicates the number of sync signals the thread `th_no` is waiting for, before getting fired. As each data unit is received, the `cnt` value is decremented, and the thread is enabled for execution when it reaches zero. The `cnt` value is reset to `rst`, and the thread is placed in the queue for scheduling.

Fig. 2.4 shows the activation tree for the Fibonacci program, while Fig. 2.5 shows a generic activation tree to illustrate that the execution of a Threaded-C program might result in the construction of an arbitrary activation tree. The rectangular blocks represent the threaded functions, and the inner circles represent the threads. The parallel function calls or the `TOKEN/INVOKE/CALL` edges (the `TIC` edge) are shown as solid arcs. The

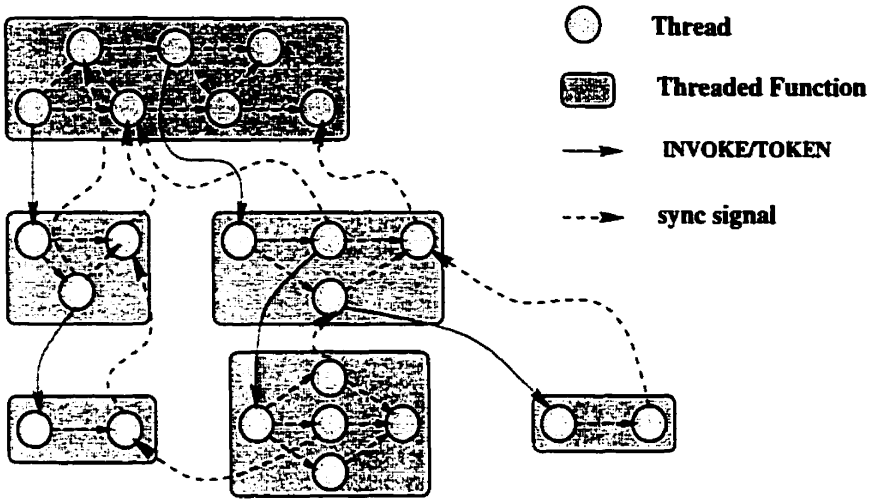


Figure 2.5: A Generic Activation tree for a Threaded-C program.

dashed arcs (or the sync edges) between different threads denote the dependencies among threads. For every dependence that is satisfied, a synchronization signal is sent to the dormant thread. The spawning of threads local to a function is depicted by the dotted arcs.

By definition, when a threaded function is invoked, its thread 0 is enabled. All the remaining threads of the function can be enabled in any arbitrary order through the manipulation of synchronization slots. Threaded-C provides for both *concurrent* (`INVOKE`, `TOKEN`) and *sequential* call mechanisms(`CALL`) for threaded functions. The syntax is described in [128]. Fig. 2.3 shows the concurrent function invocation in the Fibonacci program. Besides the `INVOKE` and `TOKEN` mechanisms, Threaded-C also provides sequential-call mechanism for threaded functions, where the thread of the the caller that executed a `CALL` primitive suspends until the callee returns. The callee function is executed immediately on the same node as the caller.

Note that Threaded-C allows for normal sequential C function calls. These functions are allocated on the stack. However, these C functions are semantically different from the threaded functions that are initiated with the sequential call mechanism (by using the `CALL` construct). The C functions are not allowed to have any Threaded-C constructs, except for the `POLL` primitive for network polling. The only similarity between the C functions and the threaded functions invoked with the `CALL` construct, is the manner in which they return. In both cases, the caller function is suspended until the callee returns.

Also, when the callee returns, execution resumes from a point after the function call.

```
THREADED vadd (SPTR done, int N, float *GLOBAL a,
               float *GLOBAL b, float *GLOBAL res)
{
   SLOT SYNC_SLOTS [2];
   int i;
   float la, lb;

   INIT_SYNC (0, 2, 2, 1);
   INIT_SYNC (1, 1, 1, 2);
   for (i = 0; i < N; i++) {
        GET_SYNC_F (a++, TO_GLOBAL (&la), 0);
        GET_SYNC_F (b++, TO_GLOBAL (&lb), 0);
END_THREAD ();

THREAD_1:
   DATA_SYNC_F (la + lb, res++, 1);
END_THREAD ();

THREAD_2:;
   }
   RSYNC (done);
   END_FUNCTION ();
}
```

Figure 2.6: Threaded-C version of Vector Addition

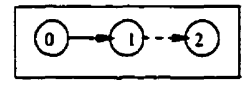


Figure 2.7: A Node in Activation Tree with a Spawn Construct

Fig. 2.7 shows the graph representation for a threaded function that uses the SPAWN construct. The solid arc denotes the spawn edge, while the dotted arc denotes a synchronization edge. The spawned thread is now in the *enabled* state and therefore ready for execution. No synchronization slots are involved in this case of explicit firing of a thread. In this figure, thread 0 spawns thread 1, and thread 1 enables thread 2 by sending it a sync signal.

In some cases, threads in a function frame may be linked by data synchronization conditions. Here, the data-synchronization is not between the *parent* and *child* threaded functions, but between the threads of the same threaded function. For example, Fig. 2.6 shows a threaded code for vector addition. Thread 1 is waiting for data from thread 0. In a way, these threads represent the producer-consumer relationship.

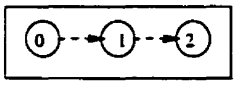


Figure 2.8: A Node in Activation Tree for Vector Addition

Fig. 2.8 shows a typical function frame for the vector addition (vadd) in one iteration. The dotted arcs represent the data synchronization edges. Another ready thread may be scheduled for execution at the thread boundary while the communication latency is being serviced. Observe that there are no TIC edges, but only sync edges.

The TIC edge may be seen as the passing of arguments from the parent node to the subcomputation represented by the child node. A TIC edge in the activation tree implies that the child node will not be invoked before the computation of its arguments. Thus we can state that Threaded-C is a *strict* language. When the sync edges from the child nodes end up only in their parent nodes, then the activation tree is *fully strict*. Threaded-C in addition, supports back-sync arcs, and any arbitrary call sequence as shown in Fig. 2.5. Efficient execution schedules with bounds on time and space are possible for strict computations [25]. The computation in Fig. 2.4 is fully strict.

There is a constraint on the TIC edge when the child threaded function is instantiated with the CALL construct. In this case, the sync edge from the child node can be destined only to the next consecutively numbered thread in the parent node which started the child computation with the CALL construct. For instance in Fig. 2.4, if the child nodes were started with the CALL construct in thread 0, then the sync edges have to return only to thread 1 in the parent node, whereas there is no such restriction with the TOKEN or the INVOKE constructs. They could have been sent to another thread, say thread 2 if desired. This is because, after the CALL construct was used to instantiate a child threaded function, the parent threaded function stalls. It resumes execution from the point after the function call, only after the child threaded function returns. However, threaded functions that are started with TOKEN or INVOKE constructs do not return any values, and therefore, can execute concurrently with the parent threaded functions. These child computations

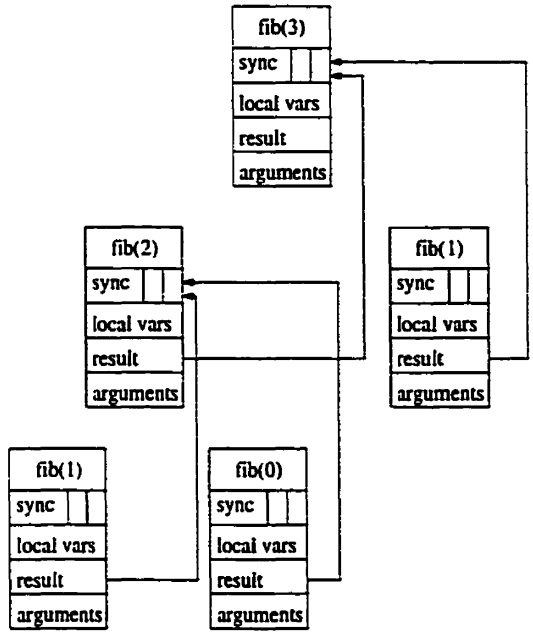


Figure 2.9: Runtime System's view with Activation Frames for fib(3)

can direct their sync edges to any thread in any threaded function, though generally they follow the *fully strict* property by sending a sync edge to some thread in their parent threaded function.

For threaded functions which are instantiated with the TOKEN construct, the host node is decided by the dynamic load balancer at runtime. The INVOKE construct specifies the node on which the function is to be executed. The thread 0 of the child threaded function is placed among other ready threads to enable execution on the node specified. The CALL construct guarantees that the child threaded function will be scheduled on the same node as the parent node, and also will be immediately executed. Also, the caller is suspended until the callee returns. The SPAWN construct spawns a local thread. If a sequential C function is started, the C function is executed on the same node as the caller function, and also returns as per normal C convention. The frames for the callee and any further nesting are allotted on the stack.

The RTS view of the Fibonacci threaded function is shown in Fig. 2.9. Thread 0 launches the subcomputations, while thread 1 is waiting for two sync signals. The *activation frame* comprises of the arguments for this threaded function, sync slots and local variables as declared in the beginning of the threaded function. The context for a threaded

function is contained in the activation frame. The sync slots help the RTS keep track of the scheduling status of those threads. All the activation frames corresponding to threaded functions are allotted on the heap, while those of sequential functions are allotted on stack. From Fig. 2.9, it can be seen that arrival of results from child computations signal the corresponding sync slot in the parent computation, which enables the associated thread for execution.

Applications with dynamic behavior, involving high levels of communication pose significant challenge to multithreaded languages. Performance studies so far have shown significant speedups with Threaded-C. This better performance is attributed to the ability of Threaded-C to handle fine-grain parallelism and the close interaction between the Threaded-C compiler and the runtime system. An efficient thread scheduling policy, and a wide choice of dynamic load balancing algorithms have helped overcome the communication/synchronization latencies.

This section summarized the model of multithreaded computations in EARTH, and prepared ground for a detailed study of the strategies required to implement a multithreaded environment in the RTS.

2.2 Preprocessing Threaded-C

This section studies the preprocessed code for a Threaded-C version of the Fibonacci presented in Fig. 2.3. The code generated contains sequential C equivalents of corresponding Threaded-C constructs. The preprocessed code for individual Threaded-C constructs in the Fibonacci program are studied. Later, at the end of this section, all these pieces are added to obtain the total translation for a threaded function. The final translation for the threaded function is listed in Fig. 2.15.

The preprocessed code for each Threaded-C construct can be divided into two parts. The first part sets the arguments for a later call to a RTS function. The second part actually calls a RTS function. It is important here that a distinction be made between the preprocessor (`etcpre`) and the preprocessed code. This section looks at the preprocessed code. Further on, the term 'function' may be read as threaded function unless specified otherwise.

Threads from the Threaded-C code are transformed into functions in the C language along with necessary linkage code. The translation sequence for Threaded-C code is

shown in Fig. 2.1. The `etcc` driver initially invokes the Earth Threaded-C preprocessor (`etcpre`) on the input Threaded-C code. The preprocessed output is C code, with threads replaced by functions and relevant calls to the RTS. The Threaded-C constructs are transformed into equivalent C code according to their semantics. The next step of the `etcc` driver is to compile the preprocesssed output to object files through a C compiler. The resulting object code is linked with the RTS object code to obtain the final executable. The `etcc` driver checks the command line options, and accordingly invokes different modules. The options for the `etcc` driver are a superset of those of any commercial C compiler. The result of a successful use of `etcc` is the final executable. The syntax of `etcc` may be observed with the '`-h`' option. The use of `etcc` is explained in section B.5. The preprocessed code for the Fibonacci threaded function is studied in this section.

A point worth mentioning before proceeding further is the allocation and disposal of heap memory. To optimize its execution and avoid frequent allocations and releases of small memory blocks, before starting the execution of an application, the runtime system reserves some heap memory into a list of available blocks, called the *free list*. Dynamic memory requirements of the runtime system during application execution, are met from the free list. When the free list becomes empty, memory is dynamically allocated using the `malloc` statement. After use, the memory is returned back to the free list. There is a free list in the address space of the RTS on every node. The contents of the free list on different nodes are not coherent.

2.2.1 Global Addresses

```
typedef struct etc_gptr {
            void *ptr;
            int node;
} etc_gptr;
```

(a) Type definition

```
_gptr1.node = etc_rts.node_id;
_gptr1.ptr = &_fp->r1;
```

(b) Preprocessed code

Figure 2.10: Type definition to represent a Global pointer and the preprocessed code for TO_GLOBAL

The EARTH model provides a global address space that allows every node to address the entire memory of the machine. EARTH constructs a global address space on distributed memory platforms by creating a *global pointer* that is formed by a node id and an address. In Threaded-C the distinction between a global and a local pointer is made

visible to the compiler [152]. Because the EARTH implementations use hardware and compiler off-the-shelf technology, the runtime system has no access to the memory management unit. Therefore global pointers are implemented in software. The data structure that stores a global address is shown in Fig. 2.10.a. To allow the use of global pointers for all standard and user defined data types, Threaded-C implements a type modifier called GLOBAL. Conversions such as TO_LOCAL and TO_GLOBAL are available in the language. Fig. 2.10.b shows the preprocessed code for the Threaded-C construct TO_GLOBAL. Local pointers are converted into global pointers by enclosing the node number and the local address as the two fields of a variable of type `etc_gptr` defined in Fig. 2.10.

2.2.2 Sync Slot

```
typedef struct etc_slot {
          int cnt;
          int rst;
          etc_handler ip;
          long fp;
} etc_slot;
```

Figure 2.11: Type definition for Sync Slot

Like global addresses, synchronization slots are implemented in software in the runtime system of EARTH. Fig. 2.11 shows the data structure used to define a synchronization slot. The `ip` field is an *instruction pointer* that contains the address of the first instruction of a thread. The `fp` field is a *frame pointer* that contains the address of the frame of the threaded function to which the thread belongs. The slot counter `cnt` contains the current value and indicates how many signals must be received before the thread associated with the slot becomes enabled and the counter `cnt` is reseted to the value in `rst`.

The key for identifying a thread is the combination called the *thread pointer* - activation frame pointer and the instruction pointer. One way of invoking a thread at runtime without knowing its name is through the function pointer. The *instruction pointer* is used to initiate thread execution by invoking the pointer to the C function representing the thread. The arguments and the context are obtained from the activation frame, which is pointed to by the frame pointer.

A thread with a non-zero sync count can be viewed as *consumer* thread (execution starts only after receipt of all relevant data). Usually the *producer* (which is another thread sending data or results of a subcomputation), after depositing the data in some global location say `result`, decrements the sync count of the consumer thread. Alternatively, the sync signal may not be preceded by any data. If the sync count reaches zero, the *enabled* thread is ready for execution.

The RTS keeps track of threads from the information contained in the sync slots. It updates the status of the thread whenever a sync signal is received at the associated sync slot. For those threads which have become *enabled*, the RTS fetches the thread pointer from the structure for the sync slot, and places the thread among other ready threads for execution.

2.2.3 SLOT_ADR

The structure for the sync slot is allocated from dynamic memory of the corresponding node. When the sync slots are to be globally accessed, the node number and the slot number form the global address of the sync slot. This is precisely the result of using the Threaded-C construct SLOT_ADR. Preprocessed code for this may be seen in the first two lines of Fig. 2.14.b. For instance, while constructing the parameters for a subcomputation, the global address of sync slot is passed as the return sync pointer 'done', as may be seen in line 6 of Fig. 2.14.b.

2.2.4 Frame based Data Structures

```
typedef struct {
          etc_slot _slots[1];
          SPTR done;
          int n;
          etc_gptr result;
          int r1, r2;
} _token_fib_F;
```

(a) Activation Frame

```
typedef struct {
          long _next, _prev, _ip, _fp;
          SPTR done;
          int n;
          etc_gptr result;
} _token_fib_P;
```

(b) Parameter Frame

Figure 2.12: Structures generated for Frame-passing

The context of a threaded function is stored in an *activation frame*. A function's activation frame contains local variables, arguments and an array of sync slots. Because Threaded-C supports parallel function invocations, various functions can be active at the

same time and terminate in arbitrary order. Thus the activation frames are stored on a tree structure in the heap, rather than on the C stack. For instance, Fig. 2.12.a shows the structure used to store the activation frame for the Fibonacci function presented in Fig. 2.3. In Fig. 2.12, `n`, `done`, and `result` are parameters passed to the threaded function `fib`; `r1`, `r2` are local variables, and the `slots[1]` array is the sync slot allocated within `fib()`.

In Threaded-C the invocation of a parallel function results in the insertion of the thread 0 of the function in the ready queue of the specified processing node. There are no assurances that the closures will be selected for execution in order. Therefore Threaded-C cannot rely on a common stack to store the parameters passed to parallel functions. A *parameter frame* is used to pass the parameters to a newly invoked threaded function. A pointer to a parameter frame structure is passed as argument to the thread 0 of every function. The parameter frame for `fib()` is shown in Fig. 2.12.b.

2.2.5 INIT_SYNC

The sync slots are initialized with the count and reset values, and are associated with a thread through the `INIT_SYNC` statement. Consider the preprocessed code for threaded macro `INIT_SYNC` in Fig. 2.14. Once the control enters a threaded function, and the activation frame is allocated on heap, the sync slot components are filled in with data pertaining to the relevant thread. This data includes current sync count, reset count, and the thread pointer.

2.2.6 SPAWN

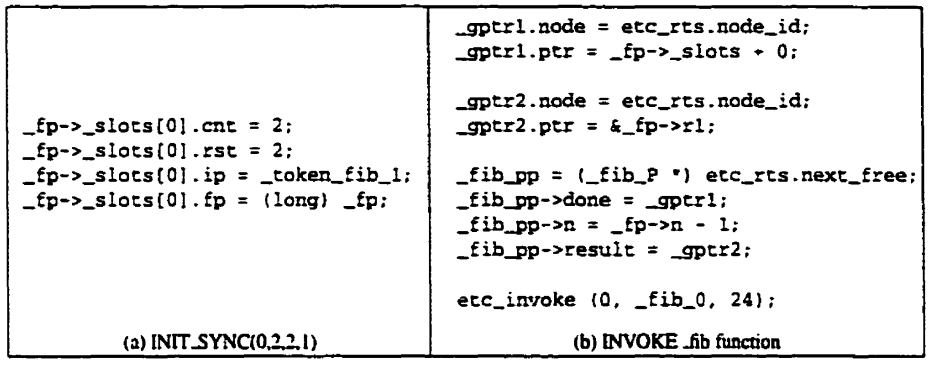
```
_gptr1.node = etc_rts.node_id;
_gptr1.ptr = _fp;
etc_spawn(_gptr1, _fib_1);
```

Figure 2.13: Preprocessed code for SPAWN(1)

The `SPAWN(1)` construct spawns locally thread 1 of the threaded function, that has the SPAWN construct. The preprocessed code is shown in Fig. 2.13. The frame pointer is converted into a global pointer. The *thread pointer* (instruction pointer and frame pointer)

is passed as an argument to the RTS function `etc_spawn`. This spawns the thread `1` in the Fibonacci threaded function.

2.2.7 INVOKE



```
_fp->_slots[0].cnt = 2;
_fp->_slots[0].rst = 2;
_fp->_slots[0].ip = _token_fib_1;
_fp->_slots[0].fp = (long) _fp;
```

(a) INIT_SYNC(0,2,2,1)

```
_gptr1.node = etc_rts.node_id;
_gptr1.ptr = _fp->_slots + 0;

_gptr2.node = etc_rts.node_id;
_gptr2.ptr = &_fp->r1;

_fib_pp = (_fib_P *) etc_rts.next_free;
_fib_pp->done = _gptr1;
_fib_pp->n = _fp->n - 1;
_fib_pp->result = _gptr2;

etc_invoke (0, _fib_0, 24);
```

(b) INVOKE _fib function

Figure 2.14: Preprocessed code for INIT_SYNC and INVOKE

The preprocessed code for the parallel construct INVOKE of the `fib()` function is shown in part (b) of Fig. 2.14. The arguments for this construct: the sync slot address and the destination location where the result is to be placed, are stored in a parameter frame structure `_fib_pp` that is allocated from the free list. The addresses for result and for the slot of the caller are converted to global addresses (`gptr1` and `gptr2`) before they are stored in the parameter frame. Once the arguments for the INVOKE construct are set up, the RTS function `etc_invoke` is invoked. The preprocessed code for the TOKEN construct is very similar, except that it does not have a processing node specification.

Fig. 2.14, part (b) shows the preprocessed code for the `INVOKE(0, _fib, SLOT_ADR(0), n - 1, TO_GLOBAL(&r1))` construct. A variable of type pointer to parameter structure (`token_fib_pp`) of the threaded function that is going to be invoked is declared at the beginning of the C function representing the first thread of the caller threaded function. Memory for this variable is allocated on top of the free list. The arguments for the child threaded function are assigned to the fields of the parameter structure variable `token_fib_pp`. Those arguments that represent addresses are converted into global pointers, and stored in two variables - `gptr1` and `gptr2`, which are then assigned to the fields of `_fib_pp`. Thus, with the arguments for the threaded function on top of the free list, the RTS function `etc_invoke` is called with the name of the first thread of the child threaded function (pointer to a C function), and the size of its

arguments.

2.2.8 Frame Passing

Each one of the threads in a threaded function is compiled into a C function. The C function representing the thread 0 of a threaded function is the first function to be executed in a given invocation and it executes only once, i.e., there are no synchronization slots associated with thread 0, and thread 0 cannot be spawned. The C function that implements thread 0 receives a pointer to the parameter frame of the threaded function. The parameter frame contains the arguments for this threaded function. The C function is responsible for building an activation frame that will be shared by all threads of the function. The activation frame shown in Fig. 2.12.a, is created in the C function for thread 0 of `fib()`. At the beginning of this C function, a request for a 64 byte buffer element from the free list is made. If no element of that size exists in the free list, then the memory is obtained by using the malloc statement[2].

Whenever a thread becomes enabled for execution, a closure representing the sequential function that contains the code for that thread is placed in a ready queue where it is scheduled for execution. This closure contains the address of the activation frame of the function that contains the enabled thread. In the preprocessed code all references to local variables or to function parameters are converted to references to fields of the activation frame structure. Thus threads belonging to the same function will share the context stored in the activation frame.

Because individual threads are transformed into C functions, thread switching is accomplished by the termination of a sequential C function and the starting of another function. Furthermore, these functions have a single parameter, the activation frame pointer. Therefore the cost to switch to another enabled thread in Threaded-C is very low.

Fig. 2.15 shows the preprocessed code for the two threads of the Fibonacci threaded function shown in Fig. 2.3. This example provides a detailed look at the *frame-passing* mechanism to share context among the threads of a threaded function.

The data structures holding the activation frame and the parameter frame of a threaded function are named with the threaded function name followed by a letter, either `'F'` or `'P'`. A pointer to the parameter frame is passed as argument to thread 0. Memory is allocated for the frame pointer in the C function for the first thread (thread 0), and its fields are

[2]The activation frame on the heap is deallocated from memory, when the `END_FUNCTION` primitive is executed.

filled with arguments for the threaded function, local variables and the sync slots. The arguments for the child threaded function are specified in the TOKEN, INVOKE or the CALL constructs. The local variables and the sync slots are obtained from the Threaded-C code for the child threaded function. The current frame pointer value is stored in the frame field (*fp*) of the sync slot that is associated with the second thread (thread 1). When this sync slot receives a sync signal that changes status of thread 1 from *enabled* state to *active* state, the RTS uses the instruction pointer to invoke the C function `_fib_1` with the frame pointer as its argument. Thus, context of a threaded function is passed between the C functions representing its threads. Observe that the threads of a threaded function are executed at different points on the time axis, though they all will execute on the same node.

2.2.9 Variable Parameter Passing

Each threaded function can have different number of parameters, and it is required to accommodate the arguments on an intermediate data structure accessible to the runtime system. The parameters for a threaded function are set up before calling the runtime system constructs for INVOKE, TOKEN or CALL. Memory is allocated on the top of the free list for the parameter structure of the invoked threaded function, and the function parameters are loaded into this structure. Using this approach we can delay the allocation of heap memory for the function until its thread 0 is activated. The runtime system function call (either `etc_invoke` or `etc_token`) receives the size of the parameter structure along with the function pointer for the starting thread of the threaded function, as shown in lines 35, 44 of Fig. 2.15.

The thread pointer is used in initiating the first thread of a threaded function, in case the threaded function is scheduled for execution on the current node. However, when a locally instantiated threaded function is scheduled for remote consumption, the runtime system sends a message that contains a pointer to the parameter structure and the address of the function to the remote node. When another node decides to execute the function, it first has to copy the parameter frame structure from the node where the function was instantiated.

```
1: void _fib_0 (_fib_P *_pp)
2: {
3:    etc_gptr _gptr1, _gptr2;
4:    _fib_P *_fp;
5:    void _fib_1 ();
6:    _fib_P *_fib_pp;
7:    void _fib_0 ();
/* Obtain memory for frame pointer */
8:    _fp = (_fib_P *) etc_rts.free_64;
9:    if (_fp)
10: etc_rts.free_64 = _fp->_slots[0].cnt;
11:   else
12: _fp = (_fib_P *) malloc (64);
13:     _fp->done = _pp->done;
14:     _fp->n = _pp->n;
15:     _fp->result = _pp->result;
/* Insert code for INIT_SYNC(0, 2, 2, 1) */
16:     _fp->_slots[0].cnt = 2;
17:     _fp->_slots[0].rst = 2;
18:     _fp->_slots[0].ip = _fib_1;
19:     _fp->_slots[0].fp = (long) _fp;
20:     if (_fp->n < 2) {
21: _fp->r1 = 1;
22: _fp->r2 = 0;
/* Insert code for SPAWN(1) */
23: _gptr1.node = etc_rts.node_id;
24: _gptr1.ptr = _fp;
25: etc_spawn (_gptr1, _fib_1);
26:    } else {
/* Globalize the local variables
   syncs slot and r1                */
27: _gptr1.node = etc_rts.node_id;
28: _gptr1.ptr = _fp->_slots + 0;
29: _gptr2.node = etc_rts.node_id;
30: _gptr2.ptr = &_fp->r1;
/* Obtain memory for Parameter structure */
31: _fib_pp = (_fib_P *) etc_rts.next_free;
/* Load parameters for INVOKE onto
   the top of free element list   */
32: _fib_pp->done = _gptr1;
33: _fib_pp->n = _fp->n - 1;
34: _fib_pp->result = _gptr2;
/* Once the arguments are set up,
   call the RTS function          */
35: etc_invoke(0, _fib_0, 24);
/* Repeat same process as above for
   next TOKEN call                */
36: _gptr1.node = etc_rts.node_id;
37: _gptr1.ptr = _fp->_slots + 0;
38: _gptr2.node = etc_rts.node_id;
39: _gptr2.ptr = &_fp->r2;
40: _fib_pp = (_fib_P *) etc_rts.next_free;
41: _fib_pp->done = _gptr1;
42: _fib_pp->n = _fp->n - 2;
43: _fib_pp->result = _gptr2;
44: etc_token (_fib_0, 24);
45:    }
/* End of code for Thread 0 */
46: }
```

(a) Thread 0

```
/* The local context of threaded function
   passed between threads through the
   frame pointer   */

1: void _fib_1 (_fib_P *_fp)
2: {
3:   etc_gptr _gptr1, _gptr2;


4:   etc_data_sync_1 (_fp->r1 + _fp->r2,
                  _fp->result, _fp->done);

/* End of code for Thread 1 */
5: _end_fun:
6:   _fp->_slots[0].cnt = etc_rts.free_64;

/* Return space for frame pointer
   to free element list          */
7:   etc_rts.free_64 = (long) _fp;
8: }
```

(b) Thread 1

Figure 2.15: Frame passing among 2 threads of Fibonacci function

2.2.10 Preprocessed Code for Fibonacci - Detailed Study

The preprocessed code for the Fibonacci threaded function from Fig. 2.3 is listed in Fig. 2.15. A typical activation tree for this code is shown in Fig. 2.4. The RTS view of this tree is depicted in Fig. 2.9.

A pointer to the parameter frame containing the arguments for the threaded function is passed as a parameter to the C function representing the first thread - `_fib_0`. The frame pointer is allocated and the relevant slots are filled in to store the local context. Next, the parameter frame structure which holds arguments for subcomputations is placed on the top of the free list. The parameters for the `INVOKE` statement are stored in the parameter structure on the top of the free list. Once the parameters are set up, the RTS function `etc_invoke` is called with the destination node for execution, instruction pointer and the size of the parameter structure stored on top of the free list. The same procedure is repeated for the `TOKEN` statement as well. The C function for the second thread in the threaded function, `_fib_1` is passed the frame pointer as parameter, thus ensuring that the second thread has access to the context of the threaded function. A line by line study of the preprocessed code for the Fibonacci follows.

line no : *Comments for preprocessed code for Thread* 0, *shown in part (a) of Fig. 2.15*

line 1: The C function corresponding to the first thread in the threaded function is `_fib_0`. The arguments to the threaded function are placed in a parameter frame structure, and a pointer to this structure is passed as argument to `_fib_0`.

line 3: A maximum of two global addresses are used in any statement in the threaded function `fib`. So, two dummy declarations for global pointers are made to accommodate remote references.

line 4: A variable is declared to hold the frame pointer. Please refer to Fig. 2.12 to view the type definition of activation and parameter frames.

line 6: A variable is declared to hold the parameter frame pointer. This holds the parameters for subcomputations (used in *variable parameter passing*).

lines 8,12: Memory for the frame pointer allocated from the free list. If memory is not available on the free list, dynamic memory is obtained by an explicit use of the `malloc` statement.

lines 13,15: The arguments to the threaded function are copied into the frame pointer structure from the parameter frame structure. Maintaining the values for arguments in the activation frame is required for context sharing.

lines 16-19: This code corresponds to the `INIT_SYNC` construct from the threaded

function fib in Fig. 2.3. The preprocessed code may be seen in part (a) of Fig. 2.14. From the parameters of the INIT_SYNC construct in fib, the values for sync count, reset count are filled into the sync slot structure for the thread in the activation frame. The thread pointer is obtained from the thread, with which this sync slot is associated.

line 20: If n<2, then send data to the global pointer in result, and decrement the sync slot done (both obtained from the activation frame).

lines 23-25: The local thread 1 (_fib_1) is to be spawned here. The frame pointer is made a global pointer, and passed along with the function pointer for _fib_1 to the RTS function.

lines 27-35: This C code corresponds to the TOKEN statement. Initially, the location for storing the result and the relevant sync slot are to be converted into global addresses. From the declaration of the frame pointer, we can see that sync slot 0 and the local variable r1 are not global pointers. Note the SLOT_ADR and the TO_GLOBAL prefixes to the sync slot 0 and the result location r1 in the TOKEN statement in the threaded function. Lines 27-30 perform this function.

line 31: The arguments for TOKEN may be any in number. An intermediate buffer area is required to hold the parameters, as its not possible to assign any specific data structure to hold the parameters, without knowing their number and type. So, the top element of the free list is type casted to the parameter frame type.

Lines 32-34 correspond to filling in the values for the newly declared parameter frame structure on top of the free list. In this case, the arguments for the TOKEN construct - the sync slot 0, the value n-1, and the result location r1, are loaded onto the top of the free list.

Now, the setup for the TOKEN construct is complete. The RTS function, etc_token is called with the instruction pointer, and the size of the parameters on top of the free list. The destination node for execution is determined by the RTS at runtime, through dynamic load balancing. The only difference between the INVOKE and TOKEN is that, the programmer decides the node for the execution of a threaded function in the case of the INVOKE.

line 36-44: The same procedure as above, is repeated for the next TOKEN construct. Note that, the destination node number is not specified in the final RTS call etc_token.

line 46: The code for _fib_0 is complete now, and the function returns.

line no: Comments for preprocessed code for Thread 1, shown in part (b) of Fig. 2.15

From the lines 16-19 in the C function _fib_0, the activation frame has the sync

slot information corresponding to thread 1. When the required number of sync signals have been received at the sync slot, the RTS obtains thread pointer for Thread 1 from the sync slot, and invokes the C function corresponding to Thread 1, by using the instruction pointer (a function pointer in C), and passing it the frame pointer as argument.

line 1: A pointer to the frame pointer, which has the local context for this threaded function, is passed as parameter to the C function representing thread 1 (`_fib_1`). This frame pointer is allocated from the dynamic memory of the node on which the threaded function is scheduled, and hence the restriction that all threads belonging to a threaded function should be scheduled on the same node.

line 4: The sum is placed in the location `result` and the relevant sync slot (`done`) decremented, within the `data_sync` function call.

line 5: This statement marks the end of code for thread 1.

line 7: The frame pointer is returned (deallocated) to the free list.

2.2.11 Sequential-Call mechanism with CALL

The discussion on frame-passing mechanism for threaded functions so far has dealt with concurrent-call mechanism for threaded functions. Its implementation for the sequential-call mechanism is slightly different.

The threaded function that calls another threaded function is the *caller*, while the threaded function that is called is the *callee*. In a sequential function call mechanism[3], the caller is suspended until the callee returns. In this case, the C function corresponding to the first thread (thread 0) of the callee has two additional parameters - the caller frame pointer and a pointer to the instruction after the CALL statement in the caller. Similarly, the last statement in the C function that represents the last executed thread of the callee, is a call to the RTS function `etc_return`. This statement is generated by the preprocessor from the Threaded-C primitive RETURN.

The preprocessed code for threaded functions with the CALL construct has at least as many C functions as the number of CALL instructions so that each CALL instruction is serviced in one C function. Each of these C functions terminates after calling the first thread of the callee with arguments. These C functions are formed in two phases. In the first phase, the arguments for the callee are stored in a parameter frame structure.

[3]Invoking child threaded functions with the CALL statement.

A pointer to the parameter frame is declared and is allocated heap memory. Some arguments, like the *result*, may have to be converted into global addresses. This parameter frame is referenced by the parameter pointer. In the second phase, the callee is invoked by calling the C function corresponding to its first thread with three arguments: the parameter pointer (to access the arguments) and the *return thread pointer*[4] for the next thread in the caller. The C function in the caller terminates immediately after initiating the callee in this manner.

The parameters for the callee and the return thread pointer are stored in the activation frame of the callee. There is no difference between the preprocessed codes for threaded functions instantiated with `CALL`, with `INVOKE`, or with `TOKEN`, except for the parameters to the callee , and its exit semantics.

The preprocessed code for a callee is shown in Fig. 2.16. Preprocessed code for a threaded MAIN function using two CALL instructions is shown in part (a), while the preprocessed code for the callee is shown in part (b). The code shown here is not complete, only the parts relevant to the discussion are included.

line no : Comments for part (a) of Fig. 2.16

line 3: The `CALL` construct is used to invoke the child threaded function `call_token_fib` with two arguments.

line 4: Another threaded function `call_mixed_fib` is the callee with two arguments.

line 7: Preprocessed code for the first thread of the `MAIN` threaded function is shown here. Though there is only one thread in the MAIN threaded function, three C functions are present in the preprocessed code, because of its two CALL statements.

line 11: A pointer is declared to parameter frame of function `call_token_fib`.

line 12: Function declaration of next C function, to which control has to be returned by the callee threaded function.

lines 18-19: The `result` location is converted into a global address.

line 20: Obtain heap memory for parameter pointer.

lines 22-23: Upload arguments into parameter frame of the callee threaded function.

line 24: A C function call to the first thread of the callee threaded function, with parameter pointer (for arguments), return instruction pointer (function pointer for next C function of MAIN), and return frame pointer (frame pointer of the caller) is made.

line 26: The first C function for the MAIN threaded function ends. The remaining

[4]Thread pointer is the combination of instruction pointer and frame pointer.

part of the MAIN code is executed in the next two C functions. This is done to enable specification of the program counter, now pointing to the statement following the CALL instruction in MAIN threaded function, as the return instruction pointer.

line 27: Control reaches here, when the callee threaded function returns, as may be seen in `line 52` of part (b) of Fig. 2.16.

lines 28-40: The same steps as above, repeated for the second CALLed function `call_mixed_fib`.

lines 41-44: This is the last C function representing the MAIN threaded function. Control reaches here after the callee `call_mixed_fib` has returned.

line no : Comments for part (b) of Fig. 2.16

We shall discuss the preprocessed code for one of the callees, the `call_token_fib`.

line 8: Declaration for return function pointer.

line 9: Declaration for return frame pointer.

line 14: This is the first thread of the callee. The C function receives as parameters, the return *thread pointer* (instruction pointer, and frame pointer) and a pointer to the parameter frame holding arguments for this child threaded function. The thread pointer points to the C function of the caller, to which control has to be returned when this threaded function exits.

lines 26-27: The thread pointer (as obtained from the parameter pointer) is stored in the activation frame of the callee.

line 45: Start of second thread of the callee. The return thread pointer is preserved in the activation frame pointed to, by frame pointer.

line 51: Returning memory held by activation frame to heap.

line 52: Returning control to C function of the caller threaded function.

The `END_FUNCTION` and `RETURN` constructs are two different ways of returning from threaded functions. END_FUNCTION is used when a threaded function is invoked with the concurrent-call mechanism (using TOKEN or INVOKE). The invoking function and the invoked function can execute simultaneously. The RETURN is used in the case of sequential-call mechanism (using CALL), when the caller is blocked until the callee returns. For this purpose, the return thread pointer in the caller is passed as one of the arguments to the callee.

```
1:THREADED MAIN (void)
2:{
        ---
3:CALL (call_token_fib, val, TO_GLOBAL (&res));
4:CALL (call_mixed_fib, val, TO_GLOBAL (&res));
5:    RETURN ();
6:}

7:void _MAIN_0 (_MAIN_P *_pp)
8:{
9:    etc_gptr _gptr1;
10:    _MAIN_F *_fp;
11:    _call_token_fib_P *_call_token_fib_pp;
12:    void _MAIN_C1 ();

13:    _fp = (_MAIN_F *) etc_rts.free_64;
14:    if (_fp)
15: etc_rts.free_64 = _fp->_slots[0].cnt;
16:    else
17: _fp = (_MAIN_F *) malloc (64);
           -------
18:    _gptr1.node = etc_rts.node_id;
19:    _gptr1.ptr = &_fp->res;
20:    _call_token_fib_pp = (_call_token_fib_P *)
21:                          etc_rts.next_free;
22:    _call_token_fib_pp->n = _fp->val;
23:    _call_token_fib_pp->result = _gptr1;
24:    _call_token_fib_0 (_call_token_fib_pp,
25:                       _MAIN_C1, _fp);
26:}

27:void _MAIN_C1 (_MAIN_F *_fp)
28:{
29:    etc_gptr _gptr1;
30:    _call_mixed_fib_P *_call_mixed_fib_pp;
31:    void _MAIN_C2 ();

32:    _gptr1.node = etc_rts.node_id;
33:    _gptr1.ptr = &_fp->res;
34:    _call_mixed_fib_pp = (_call_mixed_fib_P *)
35:                          etc_rts.next_free;
36:    _call_mixed_fib_pp->n = _fp->val;
37:    _call_mixed_fib_pp->result = _gptr1;
38:    _call_mixed_fib_0 (_call_mixed_fib_pp,
39:                       _MAIN_C2, _fp);
40:}
41:void _MAIN_C2 (_MAIN_F *_fp)
42:{
43:    etc_gptr _gptr1;

        ----
        ----
44:}
```

(a) MAIN threaded function

```
1:typedef struct {
2:  long _next, _prev, _ip, _fp;
3:  int n;
4:  etc_gptr result;
5:} _call_token_fib_P;
6:typedef struct {
7:  etc_slot _slots[1];
8:  void (*_ret_ip) ();
9:  void *_ret_fp;
10: int n;
11: etc_gptr result;
12: int r;
13:}_call_token_fib_F;

14:void _call_token_fib_0 (
       _call_token_fib_P *_pp,
15:    void _ret_ip (), void *_ret_fp )
16:{
17: etc_gptr _gptr1, _gptr2;
18: _call_token_fib_F *_fp;
19  void _call_token_fib_1 ();
20: _token_fib_P *_token_fib_pp;

21: _fp = (_call_token_fib_F *)
                          etc_rts.free_64;
22: if (_fp)
23:   etc_rts.free_64 = _fp->_slots[0].cnt;
24: else
25:   _fp = (_call_token_fib_F *) malloc (64);
26:   _fp->_ret_ip = _ret_ip;
27:   _fp->_ret_fp = _ret_fp;
28:   _fp->n = _pp->n;
29:   _fp->result = _pp->result;
30:   _fp->_slots[0].cnt = 1;
31:   _fp->_slots[0].rst = 1;
32:   _fp->_slots[0].ip = _call_token_fib_1;
33:   _fp->_slots[0].fp = (long) _fp;
34:   _gptr1.node = etc_rts.node_id;
35:   _gptr1.ptr = _fp->_slots + 0;
36:   _gptr2.node = etc_rts.node_id;
37:   _gptr2.ptr = &_fp->r;
38:   _token_fib_pp = (_token_fib_P *)
39:                     etc_rts.next_free;
40:   _token_fib_pp->done = _gptr1;
41:   _token_fib_pp->n = _fp->n;
42:   _token_fib_pp->result = _gptr2;
43:   etc_invoke (0, _token_fib_0, 24);
44:}

45:void _call_token_fib_1 (
                _call_token_fib_F *_fp )
46:{
47:   etc_gptr _gptr1, _gptr2;

48:   *(int *) _fp->result.ptr = _fp->r;
49:_end_fun:
50:   _fp->_slots[0].cnt = etc_rts.free_64;
51:   etc_rts.free_64 = (long) _fp;
52:   etc_return (_fp->_ret_ip, _fp->_ret_fp);
53:}
```

(b) Callee threaded function call_token_fib

Figure 2.16: Pre-processing of CALL instruction

2.2.12 Loops spread over Threads

Threaded-C allows for loops to be spread over more than one thread. In the preprocessed code, the loop will be represented in all the C representations of the threads. However, control does not toggle between these C functions, as it does among the threads. A threaded function, with a long *while* loop spread among 2 threads is shown in part (a) of Fig. 2.17, and its preprocessed code is shown in part (b) of Fig. 2.17.

Before proceeding further, we define the term *ready queue*. Threads guaranteed to be scheduled on a node are placed in the ready queue (RQ) of that node. Threads from the ready queue are executed in the FIFO fashion.

For instance, the `while` loop is spread over threads `0` and `1`, in part (a) of Fig. 2.17. The loop ends in the second thread (*line 13*). At the threaded function level, the understanding of execution of this loop is that, when thread `0` finishes, control enters thread `1` (as this thread is spawned from thread `0`), and after executing the only statement before the ending brace for while loop, control jumps back to the beginning of the loop body in thread `0`. This continues until the termination of the loop. However in the preprocessed code for thread `0` (`_get_loops_p1_0`) seen in part (b) of Fig. 2.17, the `while` statement is replaced by an `if` statement (*line 14*). Part of the loop body, that is in thread `0`, is mapped into the C function representing that thread, i.e. part of the first iteration of the loop is retained in this C function.

In the C function representing the second thread, the complete `while` loop is retained, including a C `return` statement (preprocessed code for `END_THREAD`), at the thread boundary (*line 42*). However, the thread `0` part of the first iteration in this `while` loop is skipped, by using a C `goto` statement (*line 34*), as that part must already have been executed in code for thread `0`.

When the thread `0` completes execution, it has spawned thread `1` (*line 17*). Control jumps to end of loop in C function corresponding to thread `1`, increments the `loops` variable (*line 44*), and jumps back to start of the `while` loop (*line 35*). The loop body is executed for next iteration, but the function terminates and exits at the `return` statement (*line 42*). Nevertheless, there is another thread `1` spawned in the loop body that shares the same activation frame. Execution of the new C function for thread `1` starts, and ends in same fashion as earlier. This continues until the loop counter reaches its limit. In the last iteration, when the loop condition fails, control jumps to the statements after the loop body (*line 46*), thus skipping the `return` statement. After the execution of the statements at end of thread `1` after the loop body, the C function terminates properly, and

```
1: THREADED get_loops_p1 (SPTR done,
2:                int *GLOBAL result)
3: {
4:    SLOT SYNC_SLOTS [1];
5:    int del, loops;

6:    loops = 0;
7:    while (!stop) {
8:       SPAWN (1);
9:       wait (10);
10:      END_THREAD ();

11:THREAD_1:

12: loops += 1;
13:  }
14:   DATA_RSYNC_L (loops, result,
                    done);
15:   stop = 0;
16:   END_FUNCTION ();
17:}
```

(a) Threaded function with `While` loop spread over threads

```
1:void _get_loops_p1_0 (_get_loops_p1_P *_pp)
2:{
3:  etc_gptr _gptr1;
4:  _get_loops_p1_F *_fp;
5:  void _get_loops_p1_1 ();

6:  _fp = (_get_loops_p1_F *) etc_rts.free_64;
7:  if (_fp)
8:    etc_rts.free_64 = _fp->_slots[0].cnt;
9:  else
10:   _fp = (_get_loops_p1_F *) malloc (64);
11:   _fp->done = _pp->done;
12:   _fp->result = _pp->result;
13:   _fp->loops = 0;
14:   if (!stop) {
15: _gptr1.node = etc_rts.node_id;
16: _gptr1.ptr = _fp;
17: etc_spawn (_gptr1, _get_loops_p1_1);
18: for (_fp->del = 0; _fp->del < 10 * 4;
19:                                _fp->del++)
20:     nop ();

21: return;
22:   }
23:   etc_data_sync_1 (_fp->loops,
                       _fp->result,
24:                    _fp->done);
25:    stop = 0;
26: _end_fun:
27: _fp->_slots[0].cnt = etc_rts.free_64;
28: etc_rts.free_64 = (long) _fp;
29:}

30:void _get_loops_p1_1(_get_loops_p1_F *_fp)
31:{
32:  etc_gptr _gptr1;
33:  void _get_loops_p1_1 ();

34:  goto THREAD_1;
35:  while (!stop) {
36:   _gptr1.node = etc_rts.node_id;
37:   _gptr1.ptr = _fp;
38:   etc_spawn (_gptr1, _get_loops_p1_1);
39:   for (_fp->del = 0; _fp->del < 10 * 4;
40:                                _fp->del++)
41:         nop ();
42:  return;

43:THREAD_1:
44:  _fp->loops += 1;
45: }
46:  etc_data_sync_1 (_fp->loops, _fp->result,
47:                                _fp->done);
48:  stop = 0;
49:_end_fun:
50:  _fp->_slots[0].cnt = etc_rts.free_64;
51:  etc_rts.free_64 = (long) _fp;
52:}
```

(b) Pre-processed code for `While` loop

Figure 2.17: Pre-processing of *While* Loop spread over Threads

exits.

One fact to be noted is that the threads spawned in each iteration will be placed at end of the ready queue, and therefore will be executed after the threads already in the queue. The early threads may correspond to another threaded function. Thus, splitting loops over threads provides a chance for threads of other threaded functions to be executed alongside/among the threads of this threaded function.

In summary, threaded code might be preprocessed into individual C functions for the following reasons:

- To separate long latency operations into separate threads, i.e., place the request for an operation in one thread, and code that uses the results of the operation in another thread. The decision to split threaded functions into individual threads is made by the Threaded-C programmer. Here each thread is represented by a C function.

- To facilitate restart of execution of a threaded function from the instruction next to the one in which a callee is instantiated with the CALL construct. By translating the remaining part of the threaded function into another C function, the C function pointer can be easily specified as instruction pointer in the return thread pointer. Here the preprocessor makes the decision as to the number and structure of the C functions to represent the threaded code.

- To enable a scheduling fashion in which, a thread can be made to execute among other threads (belonging to another threaded function), that are ahead in the ready queue. Here, the loop is designed by the Threaded-C programmer, while the preprocessor generates a C function for each thread. The structure of these individual C functions allows for the representation of a loop spread over multiple threads.

To elaborate the third point further, some applications require that a thread be scheduled on a remote node, while there is another thread (say *Th. A*) that is still executing there, i.e. scheduling a thread (say *Th. B*) on a remote node, while another thread is, for example, executing a long loop there. However, threads in EARTH are fine-grain, and non-preemptive. In such a case, the remote thread (*Th. A*) will be executing forever, and the later threads will never get CPU time. If, instead, the long loop in the remote thread (*Th. A*) is spread over two threads, and the first thread spawns the second thread, as shown in Fig. 2.17, then the second thread will be placed behind *Th.B* in the RQ, thus enabling *Th. B* to execute among the threads of another threaded function scheduled earlier on the RQ.

2.3 The Runtime System

The **Runtime System (RTS)** provides a multithreaded environment for running application threads efficiently. Its core responsibilities are thread-scheduling, context switching, data communication, synchronization, global memory management and dynamic load balancing.

Before proceeding further, it is important to study the representation of the threaded function in the runtime system. A *token* is the runtime system handle to execute a threaded function. A token consists of two parts: the name of the C function corresponding to the thread 0 of the threaded function (instruction pointer), and the amount of memory, measured in bytes, required to store the arguments for the threaded function. Tokens are created with the TOKEN construct and are the units used for dynamic load balancing.

Threaded functions that are instantiated by a TOKEN statement can be executed in any processing node, while a function instantiation initiated by an INVOKE statement must be executed in the node specified in the first parameter of the INVOKE statement.

When a token is created, heap memory for the activation frame has not yet been allocated and hence the tokens are free to migrate to remote nodes. Once the threaded function is scheduled for execution on a node, i.e. guaranteed to execute on a node, memory is allocated for the frame pointer on that node during the first thread of the threaded function, and the arguments for the threaded function are down-loaded into the parameter frame structure. The frame now contains the context of the threaded function, and can be passed onto remaining threads of the threaded function. Thus the activation frame for a threaded function is *expanded*. Once the activation frame has been expanded, the threaded function cannot migrate to another node. Also, all the threads of the threaded function have to execute on the same node.

Though this document details the SP-2/tb-3 version of RTS, the operating principles are the same among all versions, barring minor implementation details that are localized to a few routines in the CPU and network interfaces.

2.3.1 Context Switching

Regarding context-switching, threads in Threaded-C are non-preemptive, therefore the term *context* here does not mean the usual combination of the register file, program

counter, stack pointer, and the status register. Instead, the context of a threaded function includes the arguments, sync slots and local variables of a threaded function. Therefore, context-switching should be seen as *context-sharing* in Threaded-C. Once the token representing an unexpanded Threaded-C function is expanded, the component threads form separately invocable C functions representing the non-preemptive threads. Context-switching between threads is made simple by terminating one C function, and starting another C function. This manner of context-switching reduces overheads as it does not require any context (register file, program counter, stack pointer, status register) to be saved. Absence of such context savings improves portability, as the RTS code does not need to access any machine specific areas. Context sharing between threads of a threaded function has been explained in section 2.2.8.

2.3.2 Scheduling of Threads

Scheduling a thread involves two important decisions: *where* the thread should be executed, and *when* it should start execution. In the EARTH architecture model, these two decisions are kept separate. It is the responsibility of the load balancer to decide where the tokens will be executed, while the EARTH scheduler decides the local thread execution order within a node.

In the EARTH model scheduling is the last action performed by the runtime system on a thread before its execution, similar to *dispatching* in process scheduling. The node that will execute a thread is identified prior to scheduling either by the compiler/programmer or by the dynamic load balancer. Load balancing is at a higher level in the RTS functional hierarchy than scheduling, as may be noticed in Fig. 2.20. Load balancing makes threads available for scheduling. While scheduling is confined to one node, load balancing happens between all nodes.

To allow the migration of tokens between nodes and to enable the implementation of dynamic load balancing, two queues - the `ready queue (RQ)` and the `token queue (TQ)`, are maintained by the runtime system on each processing node. Whenever a thread becomes enabled it is inserted in the RQ. An `INVOKE` causes the thread 0 of the invoked function to be placed in the RQ while a `TOKEN` causes a token representing the token-ed function to be placed in the TQ. Load balancing operates on the TQ, whereas scheduling operates on the RQ.

Threads, rather than tokens, are the units of work in the RQ. These threads are the

components of the threaded functions that are guaranteed to execute on a node. The threads in the RQ are in the *enabled* state since all of their synchronization conditions have been met. While threads of a threaded function may execute in consecutive order, this order is not guaranteed as the firing of a thread depends on its synchronization conditions. Remember that all the enabled threads of a threaded function have to execute on the same node. The FIFO scheduling policy is used to execute threads in the RQ. Threads are added at the tail, and removed from the head of the queue. Whenever the RQ is empty, a token is fetched from the TQ and the instruction pointer from the token is used to launch the first thread of the threaded function.

In contrast, the TQ in contrast is a DEQUE - a data structure similar to a queue, but operatable on both ends. Tokens are the units of work in the TQ. The TQ behaves locally like a stack. When a node generates a token, the token is appended to the tail of the TQ (PUSH operation). For local consumption, a token is extracted from the tail (POP operation). However, the TQ acts like a FIFO queue when a token is to be sent to remote nodes as part of a load balancing operation. A token is removed from the head of the TQ, and sent to a remote node. When remotely generated tokens are received at a node, they are added at the head of the TQ. The basic principle is that, tokens are removed from the tail of the TQ for local consumption, and from the head of the TQ for remote consumption. The flow of tokens amongst the application, RQ, TQ and remote nodes is shown in Fig. 2.18.

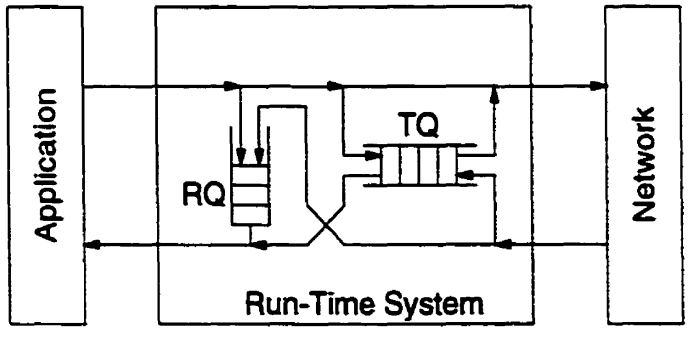


Figure 2.18: Internal Queues in the EARTH RTS

The TQ design is intended to reduce token migrations and thereby reduce space explosion (memory space for holding the activation frames). When expanding tokens or exchanging them with other nodes, the choice of a token is important as it determines the order in which the program is executed. The execution order in turn determines the

amount of parallelism which can be exploited and the amount of memory needed to execute the program [109, 107].

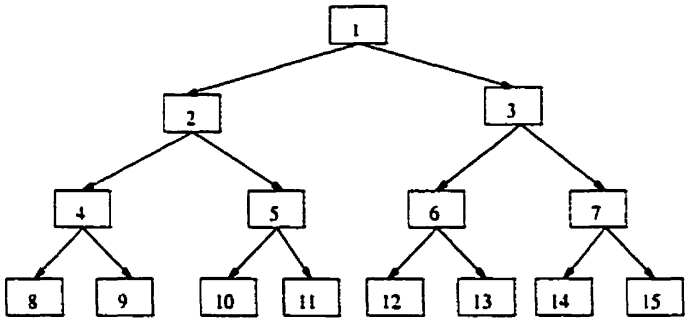


Figure 2.19: A Sample Activation Tree

Fig. 2.19 shows the activation tree of a simple, doubly recursive function. The rectangular blocks represent threaded functions, while the edges represent the TIC edges. If both calls can be executed in parallel, several execution sequences are possible. For instance, the sequence 1, (2,3), (4, 5, 6, 7), (8, 9, 10, 11, 12, 13, 14, 15) corresponds to a *breadth-first* execution. This execution order makes it possible to execute threaded functions 8 to 15 in parallel, i.e. maximum parallelism of 8. On the other hand, the sequence 1, 2, 4, 8, 9, 10, 11, 3, 6, 12, 13, 7, 14, 15 corresponds to a *depth-first* execution. In that case all threaded functions are executed sequentially.

The main differences between these two execution orders are the amount of parallelism that can be exploited and the amount of memory needed to execute the program. The depth-first strategy, which is the normal execution strategy for sequential processors, does not exploit parallelism, but at most 4 instances of the threaded function are active at the same time. On the other hand, the breadth-first approach takes advantage of all available parallelism but there must be enough memory to keep all 15 activation frames in memory at the same time. The choice of the execution order depends on the number of nodes in the multiprocessor system, , the time needed to start a threaded function on another node, the amount of work available to the other nodes, etc. An optimum strategy would be to exploit just enough parallelism to keep all processors busy while at the same time minimizing memory usage.

The TQ design approximates the ideal behavior, as explained above, quite well. For instance, when all processors are busy, no tokens are exchanged over the network. The TQ acts like a stack, and the token that was generated last is the first to be expanded. This

corresponds to the normal depth-first execution strategy of sequential processors. On the other hand, when a node sends the tokens it has produced to one of its neighbors, it is the oldest token which is sent first. The resulting execution order is breadth-first, as the threaded functions nearest to the root of the activation tree are executed first.

The execution order therefore becomes dependent on the load of the machine. As long as the machine is busy, the code will be executed mostly in a depth-first order, using minimum of resources. As soon as one of the processors needs more work, however, tokens are sent to it in breadth-first order and some more parallelism is exploited.

Another advantage of this strategy is that it is able to take some advantage of locality. Executing code depth-first means that threaded functions are executed on the same node as their parent. Because parameters and results can be created and accessed locally, network traffic is reduced. This works especially well in the case of recursive functions.

To summarize, depth-first expansion of the activation tree is desired locally, whereas breadth-first expansion is preferred over a set of nodes. Function frames at higher levels in the activation tree represent more work than those in lower levels. Therefore, frames with more work in the activation tree should migrate to remote nodes, to offset the work done on remote nodes with the migration costs. Depth-first expansion on a node not only reduces token migrations, but also adds to the locality of the tokens migrated. When this idea is mapped on to queue structure, the tokens near the head of the TQ correspond to the functions on the top level of the activation tree. Hence the TQ is accessed in FIFO fashion for remote consumption, and in stack fashion for local consumption. The effect of a DEQUE structure for token queue is described in [109, 107] and [33].

2.3.3 Thread Execution by the Runtime System

The multi-threading support provided by the runtime system includes updating thread's state according to the synchronization operations performed, scheduling enabled threads for execution, polling the network at thread boundaries for messages, performing dynamic load balancing, and sending data and synchronization signals to remote nodes. The RTS keeps track of tokens produced in the token queue, ready threads in ready queue, and initiates the threads by accessing their C function pointers.

During initialization, the executable of a Threaded-C program is loaded into memory of all nodes participating in the execution. Remember that the executable includes object code for both the RTS and the application. Thread launching starts on node 0 from the

'_MAIN_0' function, which corresponds to the first thread of the MAIN function of the Threaded-C program. Child threaded functions are initiated by adding tokens to the TQ on the relevant node. Enabled threads are placed in the RQ for execution. In the case of idle nodes, the dynamic load balancer on that node is invoked and it fetches tokens from other nodes according to the load balancing policy. Eventually threads are running on all nodes. The execution ends when no more threads remain to be executed.

When the execution of a thread completes the runtime system polls the network to check for any incoming messages. After processing all the incoming messages, it calls the load balancer algorithm. The normal conditions for the load are established by the load balancer algorithm. The load is abnormal if the node has less tokens than a minimal threshold or more tokens than a maximum threshold. If the load is normal, the load balancer returns immediately and the RTS fetches the next thread in the ready queue and starts to execute it. If the RQ is empty, the RTS fetches a token from the token queue. If the TQ is also empty, the load balancer on the node is invoked to request remote workload. The choice of the remote node is based on the load balancer policy. All load balancer related activities like: token request; token response; token forwarding; and any updation of load state information, happen at the thread boundary. The RTS activity [33] at a thread boundary is depicted in Fig. 2.20.

An incoming message is in one of the four groups :

Sync Request : For a sync operation, the message contains the address of a synchronization slot. The sync count is decremented, and if it becomes zero, the sync count is reset and the relevant thread is placed in the ready queue for execution. The handler routine is `etc_sync`.

Data Request : In the case of a request from a remote node for local data, the address of the first location and the size of the data block requested, and the global address of the destination are in the message. The local data is composed into a message and sent to the requesting node. The handler routine is `etc_get_sync_X`.

Response to Data Request : When remote data is received following a request sent earlier (`etc_get_sync_X`), the message contains the data, the destination address and the address of a sync slot that is to be synchronized to signal the arrival of the data. The data is placed in the local destination address, and the sync count decremented. The sync count is accessed from the sync slot that is specified in the message. The handler routine is `etc_data_sync_X`.

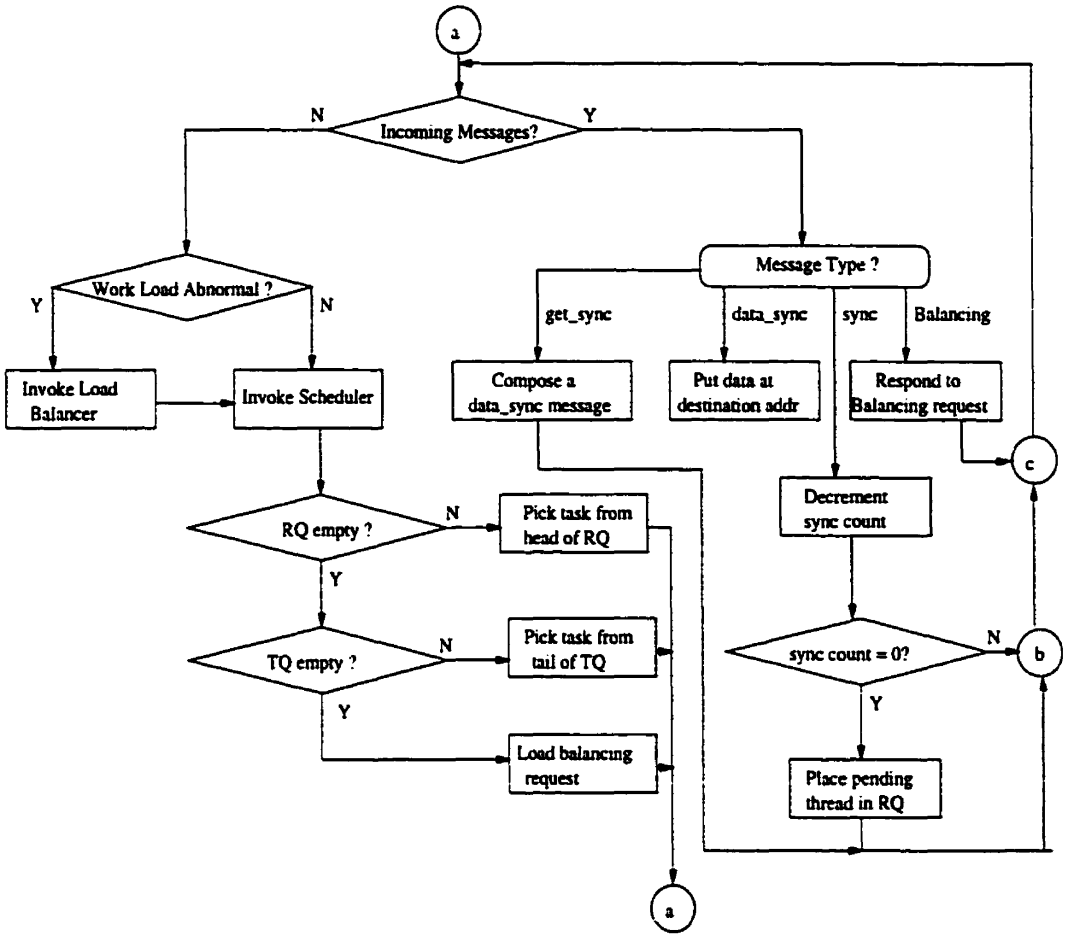


Figure 2.20: RTS activity at Polling

Load Balancing Request : A load-balancing request is serviced depending on the load-balancing policy. In general a node that has few tokens will store an arriving token on its token queue and reject a request for token. A node that has a lot of tokens will reply to requests with tokens and will reject new tokens. Otherwise, the request is forwarded to the next logical neighbor. The handler routine is `hdl_token_req`.

Response to load balancing request : The incoming message contains a token from a remote node. This token is consumed if the node is still idle. Otherwise, depending on the load balancer policy, this extra token is either appended to the tail of the TQ, or forwarded to next neighbor. The handler routine that performs this operation is `hdl_token`.

When data requests are received, the destination to where the data is to be sent is specified as a global address. Likewise the synchronization slot that is to be synchronized

when a data transfer is completed is specified by a global address. The runtime system reads the node portion of these addresses to decide about where to send the data reply or the synchronization signal. For efficiency, the RTS implements a number of specialized "handler" functions to process each kind of message. This multiple handlers system prevents unnecessary function invocations when a message arrives.

2.3.4 Dynamic Load Balancing

Dynamic load balancers have been well studied for coarse-grain parallel distributed computing. However, the balancer overheads in such systems are not permissible in fine-grain systems like EARTH where a token can take as little as 2 μs to run. In such systems the load balancer overheads must be kept to a minimum.

The load balancing goal in EARTH is *to ensure that all nodes are busy* rather than to balance the tokens equally among all the nodes. A node is *idle* when it has no threads to execute, while a node with surplus workload is *rich*. The balancers are implemented in a distributed manner, i.e. any load distribution information is kept by each node and there is no central authority to distribute the load. The action of individual load balancers must, over time, ensure that most of the nodes are busy when there is enough parallelism available in the application. The balancer on each node is invoked at every thread boundary whenever tokens are to be sent or received. Tokens, the units of workload, are stored on the token queue on all nodes. Migratable tokens might be produced locally or they might arrive at the node as a result of load balancing requests sent when the node was idle.

The initial version of the portable EARTH runtime system supported seven dynamic load balancer policies [33, 34]. The goal is to design simple balancers that deliver good load distribution with minimum overheads. A virtual ring network topology is adopted in all the balancers with nodes numbered clock-wise. The balancing activities might be initiated by an idle node that wants work, called a *receiver-initiated* balancer, or by a rich node that wants to distribute some of its extra work, called a *sender-initiated* balancer; or it might be initiated by either type of node, called a *hybrid* balancer.

Two types of messages are exchanged among the nodes to implement dynamic load balancing:

Load balancing request : A load-balancing request is serviced according to the load-balancing policy. In general, if there are surplus tokens in the token queue, a token is sent to the remote node that requested work. Otherwise, the request is forwarded

to the next logical neighbor in the ring.

Response to load balancing request : The incoming message contains a token from a remote node. If the node is still idle, it consumes the token. Otherwise, the token might be inserted in the head of the TQ or it might be forwarded to the next neighbor in the ring. The action taken in this case depends on the load balancer policy.

We now present a brief description of the policy adopted by each load balancer.

Dual In this load balancer a request does not contain the identification of the node that originated it. When a node is idle it creates a request and sends it to its previous neighbor in the ring. When a rich node receives a request it sends a token from its TQ to its successor. When an idle node receives a request it forwards the request to its predecessor. When a rich node receives a token, it forwards the token to its successor. When an idle node receives a token it schedules the token for execution. Notice that requests will circulate counter-clockwise in the ring while tokens will circulate in a clockwise fashion.

In EARTH implementations in which there is a single processor in each node the processor has to perform the functionality of both the execution unit and the synchronization unit. In such an implementation the dual load balancer might result in poor performance because the processor of busy nodes might be overloaded with the passing of tokens and requests. However the dual balancer can work quite well in implementations in which there is a separate processor to implement the functionality of the synchronization unit.

Spn This balancer is similar to the Dual balancer, except that it is designed for single processor nodes. To reduce the communication traffic, a request contains the identification of the node that originated it. Requests are still circulated around the ring in a counter-clockwise fashion. However, when they arrive at a rich node, the node sends the reply directly to the node that has originated the request. This policy eliminates the work that intermediate nodes would have to do to forward tokens.

Shis In the Shis balancer the rich node that sent a token in response to a reply attaches its own id to the token. The idle node that receives the token "remembers" where it comes from and directs its request to the rich node that sent work the last time. If now that node is no longer rich, the request is circulated around the ring in a

counter-clockwise fashion by the formerly rich node. This policy eliminates the overhead of forwarding tokens and reduces the overhead of forwarding requests.

Snd The sender balancer starts working when the number of tokens in the TQ reaches a threshold. The tokens are sent around the ring in a round-robin fashion to avoid overloading other nodes. If a rich node receives a token it will forward it according to its own round-robin sequencing. If a token arrives back at the node that first sent it out, it is no longer sent out. Under this policy an idle node does not attempt to balance the load. It stays idle until it receives work from a rich node.

His This history balancer employs both receiver and sender initiated strategies to distribute load. Each node keeps a list of nodes that are likely to be idle and of nodes that are likely to be rich. Such list is compiled from the recent history of load balancing activities. Nodes that sent request for work recently are likely to be idle. Nodes that replied to requests for work providing tokens are likely to be rich. When the TQ becomes empty, the history balancer sends requests to the nodes that are likely to be rich. When its TQ becomes larger than a given threshold, it sends tokens to the nodes that are likely to be idle.

Range and Catapult Requests are sent to the predecessor node as in the dual and spn balancers. When a token request reaches a node, it implies that all the nodes in the ring between the node that originated the request and the node that received it are idle. A *range list* is compiled to store the ranges of nodes that are idle. When the TQ of a node surpasses a given threshold the node will send tokens to the nodes in the range list. The catapult balancer sends tokens to the near end of the range list, whereas the range balancer sends them to the far end of the range list.

If a token reaches a node that is no more idle, the range balancer on that node forwards the token to its predecessor and sends a message to the source node to notify that it is no longer idle. The source node then updates its range list.

Fig. 2.21 shows the virtual ring and the functionality of receiver-initiated, sender-initiated and hybrid balancers in the EARTH runtime system. The seven load balancer policies provide a platform to study the various parameters that affect application performance, like the application model, grain size, logical topology, polling interval, and scalability. Our experimental results so far, show that there is no perfect load balancer for

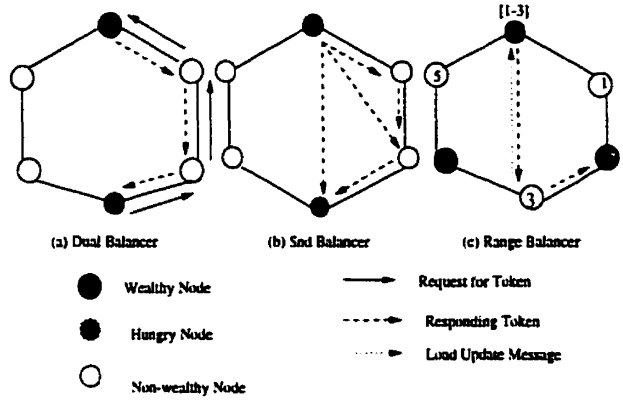


Figure 2.21: The *dual*, *snd* and *range* load balancers

all application models. However, hybrid balancers that are based on history information performed well in most situations.

2.3.5 Network Layer

The services provided by the RTS that require inter-node data communication are based on the technique of active messages [163]. An *active message* contains data and a pointer to a function that is to be invoked in the destination node when the message is received. Remote operations involving spawning of threads, sync operations, handling global memory, and inter-node data communication are based on the technique of active messages. For efficiency, and to isolate the interactions with the network from the rest of the RTS, a limited set of functions is used for inter-node communication.

The arrival of an active message causes a function handler to be invoked at the destination which acts on the data transmitted. The message transmitted contains two parts: the name of the handler routine to be invoked at the remote node, and the parameters for this handler routine. The C function pointer is used to represent the starting address of the handler routine on the remote node.

The data packets are identified depending on the number and size of the parameters transmitted as part of the active message. The RTS provides four primitives for sending active messages. All data packets have the handler routine name, but differ in the number of parameters they support. The handler routine at the remote node is invoked with these parameters.

- `etc_send2`: For data packets with two parameters.

- `etc_send4`: For data packets with four parameters.
- `etc_send6`: For data packets with six parameters.
- `etc_send2n`: For data packets with two parameters and a data block.
- `etc_send4n`: For data packets with four parameters and a data block.

These five prototypes are sufficient to achieve inter-node communication for all types of data. They are part of the network interface of the RTS, and access the network card data structures (the tb-3 card data structures on the IBM SP-2 platform). To maintain speedy communications, the network card data structures are retained in the RTS communication interface, but this interaction is very limited to a few routines. By maintaining the above prototypes for the `etc_send` routines, the rest of the RTS code is independent of the network platform on the underlying parallel machine.

The number of tasks and the identity of the current node is obtained from the network environment. The network parameters like pointers for the send, receive, and overflow buffers are initialized in the network card data structures. Acknowledgment messages are sent so that remote nodes can send some more packets. When the send queue is full, outgoing messages are shifted to the overflow buffer, from where they are later sent.

Active messages are sent through the `etc_sendX` routines. The data packet is loaded into the tb-3 network data structure along with the number of parameters required for executing the handler routines at the remote end - either 2, 4, or 6. The composed data packet is placed in the network send queue. Fig. 2.22 shows the prototypes for the send routines.

Polling the network is performed at the thread boundary, or at the explicit use of the 'POLL' statement [108]. The incoming message is picked from the receive queue, and depending on the message type (number of parameters), the handler routine is invoked with the parameters (both obtained from the message).

The RTS makes use of the `etc_send` primitives to implement remote data communication, remote synchronization, global memory management, and dynamic load balancing. The handler routines are present on all nodes, as the same executable is running on all the nodes. For instance to implement remote synchronization, the RTS on the local node composes an active message with the handler routine name (`etc_sync`) and the global address of the sync slot. The handler routine on invocation at the remote node checks for node identity, and then performs the synchronization operation. While dealing

```
void etc_send2(int dest,etc_handler hdl,int p1,
               int p2)

void etc_send4 (int dest etc_handler hdl,int p1,
                int p2,int p3,int p4)

void etc_send6 (int dest,etc_handler hdl,int p1,
                int p2,int p3,int p4,int p5,int p6)

void etc_send2n (int dest,etc_handler hdl,int p1,
                 int p2,void *p,long bytes)

void etc_send4n (int dest,etc_handler hdl,int p1,
                 int p2,int p3,int p4,void *p,
                 long bytes)
```

Figure 2.22: Send routines for Active Messages

with global addresses such as that of the sync slot in this example, the handler routine first checks the node number in the global address. If the node identity number matches the node number in the global address, then the requested operation is performed, otherwise the message is forwarded through an `etc_sendX` routine to the destination node. Thus global addresses are supported over the network layer.

2.3.6 Common RTS core

This section looks at the organization of the RTS source code while implementing some of the RTS core functionality - synchronization and scheduling, data communication and global memory management.

The starting point for the executable is the 'main' function in the file `rts.c`. The RTS variables, timer, network, load balancer, and profiler (if load balancer and profiler are chosen as options on `etcc` command-line) are initialized. The `etc_run()` function starts the execution of the Threaded-C code from it's `_MAIN_0` function and from here on, the RTS keeps track of application execution. The `etc_next_thread` function specified in the load balancer modules, determines the next thread to be executed - either from the RQ, TQ, or if both are empty, by requesting from the logical neighbors as per the load balancing policy. The threads are placed in the RQ by placing the function pointer

in the 'ip' field, and the parameters in the 'fp' field.

The file `data.c` contains the macro names for the handlers. The inline expanded macro definitions are given in the file `data_inc.c`. The handlers are organized in two levels. The first level corresponds to the handler name obtained from the incoming message at polling. These invoked handlers have their name starting with the keyword `etc_` (for ex. `etc_sync`). The function of `etc_XXX` is to determine if the message is meant for the current node. If the message is for the local node, the corresponding function whose name starts with the keyword `hdl_` is invoked. The routine `hdl_XXX` proceeds to act on the parameters and perform relevant function. On the other hand, if `etc_XXX` determines that the incoming message is not meant for the local node, it forwards the handler name and parameters to the destination node. Fig. 2.23 gives an example of invoking handlers with the code for **etc_sync**. The code for `hdl_sync` is shown in Fig. 2.24.

```
/*
void inline_etc_sync (s_node, s_sp)
*/
#define inline_etc_sync(s_node, s_sp)  \
{\
 buf_elem *bp; \
 \
 if (s_node == etc_rts.node_id) { \
     inline_hdl_sync (((etc_slot *) s_sp)); \
 } else { \
 etc_send2 (s_node, hdl_sync, (int) s_sp, 0); \
 INC_REMOTE \
        } \
}
```

Figure 2.23: Invoking handler for Sync Operation

Global memory management makes use of the handlers as specified above. For a global address, the RTS on a node sends a request to a remote node which hosts the global address. The destination node, on identifying the node number in the global address as its own, immediately composes an active message with the data and the name of handler routine and sends it to the requesting node. On the other hand, if the node number in the global address does not match the node identity, the RTS forwards the message to the correct destination. This strategy of checking for the node number before performing an

operation, not only provides fault-tolerance from missing messages, but also enables the RTS code to run on any interconnection network in the parallel machine.

To access a global address, the requesting node composes an active message with the global address, and the handler routine `etc_get_sync_X`. This message is sent to the node which hosts the global address. When the host node receives a `get_sync`, it constructs a `etc_data_sync_X` message which after returning the data content, also decrements the sync counter for the relevant thread. If the sync count is zero, the thread is placed in the RQ.

Regarding thread synchronization, an explicit sync message causes the handler routine `etc_sync` to be invoked on the destination node. The handler routine decrements the sync counter, and if the sync count reaches zero places the corresponding thread in the RQ. Fig. 2.24 shows a typical handler for sync operation.

```
#define inline_hdl_sync(sp) \
{\
 buf_elem *bp; \
 \
 if (sp->cnt == 1) {\
     sp->cnt = sp->rst;\
     bp = etc_rts.next_free;\
     etc_rts.rdy_t->next = bp;\
     etc_rts.rdy_t = bp;\
     etc_rts.next_free = bp->next;\
     bp->ip = sp->ip;\
     bp->fp = sp->fp;\
     if (!etc_rts.next_free)
          etc_alloc_buf_elem ();\
 } else\
 sp->cnt -= 1;\
}
```

Figure 2.24: Handler for Sync operation

2.3.7 Architecture Specific Code

Instruction timing and cache parameters directly influence the execution time. Timing statistics for the instructions are obtained by accessing the on-board programmable timer. Assembly language routines for the IBM SP platforms return the accurate real-time clock,

after accessing the on-board 64 bit *time base* register. These routines are called from the C function, `etc_time()` whenever system time is required. The `etc_time` routine returns time in *seconds*, as a double-precision value.

In the early versions of the portable RTS, the `ct_read()` routine is used to obtain timing information. This routine was used to maintain compatibility with the Threaded-C code developed for the EARTH-MANNA. The `ct_read()` routine in turn called `etc_time_raw()`, which is in assembly, and returned time in nanoseconds as a double precision value. But since the current version of the portable RTS, `etc_time()` is the routine used to access on-board real time clock, and should be used in future Threaded-C applications and the RTS profiling code.

To obtain exact execution times, cache misses are reduced by touching both the code and data segments initially. Corresponding assembly language routines flush cache lines specified as a single address or in an address range. In addition, memory-to-memory copy operation optimized for small blocks using both integer and floating-point registers is provided. Time delay is provided by an assembly language routine `etc_delay()`.

The above stated functions contribute to the minimal machine specific code of the portable RTS. These routines are available for RS6000 CPU, and SUN workstations respectively.

2.3.8 Portability

Two features that affect the portability of an implementation of a parallel programming model are the interfaces with the *hardware* and with the *interconnection network*. The EARTH runtime system is written in a standard programming language (in this case ANSI-C) that is supported by most parallel machines. The EARTH runtime system minimizes specific references to architecture or hardware features and network protocols. To further enhance portability, specific tasks required to interact with a given network are written as a separate set of functions with a clearly defined interface with the rest of the runtime system.

When threads can be preempted the context in which a thread is executing -program counter, status registers, stack pointers and architectural registers, must be saved when the thread loses the CPU. Because threads run to completion in EARTH, context only has to be saved at the thread boundaries. At that point there is no need to save the values stored in the architectural register, program counter stack pointer or status register. The *context*

here, means the combination of the parameters, local variables, and the sync slots for a threaded function. All the context that is shared among EARTH threads belonging to the same threaded function is stored in the local variables of the function and thus is stored in the activation frame. Because there are no architectural dependent values to be saved and restored during the execution of EARTH, the runtime system can be made very portable.

The current version of the runtime system is ported onto the SUN workstations, IBM SP-2, and the Beowulf. Its network interfaces maybe myrinet, tb-2, tb-3, and TCP/IP. The RTS can be ported onto new platforms very easily and quickly.

Chapter 3

Dynamic Load Balancers in the EARTH Runtime System

Dynamic load balancing in fine-grain multithreaded systems places the following demands on a load balancer: accuracy, inexpensiveness, quick response time, simplicity, stability, efficiency, and scalability. An efficient balancer provides agreeable performance improvement with load balancing, when compared to a "no-load balancer" situation. These demands are understandable in the EARTH system, where average grain size can be 200 μs (approx. 12500 cycles). Another important expectation from a load balancer is that it should perform well for all applications, in high/low load situations. In other words, the balancer should be able to exploit the available parallelism in any application at runtime, and minimize the idle times by keeping all the processors busy.

To meet all the demands enumerated above, a balancer should not only be able to respond to rapidly fluctuating loads, but also be able to make accurate decisions based on global system state, and also consume minimum CPU cycles for load balancing purposes. As a study in this direction, the *Rand* balancer is presented in this chapter. This balancer is based on a randomizing algorithm, performs the roles of both sender and receiver according to the current load situation, considers global load state before choosing a target node for work transfer, and uses a completely connected graph as a logical topology between all the nodes in the execution.

Before proceeding further, certain assumptions in the load model are restated here. Firstly, scheduling is separated from the task of dynamic load balancing in the EARTH system. As explained in section 2.3.2, scheduling decides on the next thread to execute from locally available work in the ready queue, whereas load balancing operates between

the token queues on different nodes. The DEQUE structure of the token queue supports locality considerations by allowing depth-first search locally, and breadth-first search remotely. Tokens are the units of dynamic load balancing. Secondly, the goal of load balancing here is to keep all the processors busy, rather than balancing workload equally on all the processors.

A node is said to be in the *idle* state when it has no threads to execute. A node with surplus workload is called a *rich* node. Threshold values for workload classify the nodes as either rich or idle. Distributing the workload during application execution is achieved by sending the *tokens* to the balancers on remote nodes. A token contains all the necessary information to create a new thread. Tokens are stored in the *token queue* on each node. The token queue is based on the DEQUE data structure, which acts as a stack for local consumption and as a FIFO queue for remote consumption. Tokens eligible for migration are obtained in two ways: by generating locally, and as a result of load balancing requests. The token execution time determines the grain size.

This chapter is organized as follows: Section 3.1 reviews previous work on randomizing algorithms. Section 3.2 describes the *Rand* balancer algorithm. Section 3.3 isolates various features of the *Rand* balancer, and studies their effect on performance. Section 3.3.4 presents the algorithm for a balancer that allows to execute parallel applications with *no* load balancing.

3.1 Background

In the random mapping strategy, initial placement of threads is done by choosing a processor at random, and map that thread to that processor [99, 18, 54]. It has been shown that this scheme has a bad worst-case behavior when a highly loaded processor is chosen [50, 78, 136]. An alternative approach is to first probe a limited number of nodes at random, and then choose the best one [117, 136]. Even probing only two nodes reduces the expected maximum load when mapping n threads from $O(\log n / \log \log n)$ to $O(\log \log n)$ [20].

Mitzenmacher *et al.* [119] improvise on the work in [20]. They consider the following dynamic model: customers arrive as a Poisson stream of rate at a collection of n servers. Each customer chooses some d servers independently and uniformly at random from the n servers, and waits for service at the one with the fewest customers. Customers are served according to the first-in-first-out (FIFO) protocol, and the service time for a customer is

distributed with mean 1. This model is called the *supermarket* model. In this model, customers arrive over time, and the number of customers is not fixed. It has been shown that the maximum load is then only $(\log \log n)/\log d + O(1)$ with high probability. This same result can be observed even in a static system, in which n balls are placed into n bins, each bin chosen independently and at random. Here, each ball is placed sequentially into the least full of d bins chosen independently and uniformly at random. In the static model, the number of balls and bins are fixed. The difference between the model considered here and the one shown in Azar *et a l* is that in the latter, there are a fixed number of customers to be distributed who never leave the system. In addition, a customer who completes service is recirculated in the system.

Dynamic task arrivals, and variable number of tasks in an execution suit the multithreaded model. However, implementing the random probes is a costly process in fine-grain systems. Executing a random function for every decision, and polling overheads due to the load probes pose the biggest challenge to make load balancing profitable in fine-grain multithreaded systems such as EARTH.

The basis for the *Rand* balancer is the supermarket model [119] in distributed computing. The resuts show that giving each ball two choices instead of just one leads to an exponential improvement in the maximum load on any node. The system considered has a very high number of queues to chose from, in the order of hundreds. While having more than a hundred processors is possible, their number is rarely more than 32, or at most 64 in actual application executions. The value for d has not been formulated in [119]. Finally, the *supermarket* model is considered for process-based distributed computing, where the queuing theory principles apply. However, in multithreaded systems, task arrival rate is not independent of task consumption rate. Furthermore in fine-grain multithreaded systems, the application grain size is very small, and the simulation results in [119] do not apply here. Therefore, the *Rand* balancer has to be very lean, and resort to various other techniques so that the balancing benefits dominate the balancer overheads.

3.2 The Rand Balancer

The main features of the *Rand* balancer are as follows:

- The balancer is hybrid (symmetric)[1]. The balancer has both sender and receiver initiated components.

- In the receiver mode, load probes are sent to randomly chosen nodes. The least loaded node is chosen as the destination node for load transfer. Load probes are not used in the sender mode of the balancer.

- Load information is collected from load probes, load messages, and piggy-backed messages. This load information is used in deciding a destination node for load transfer. Care is taken to avoid *aging* of the load information. If the load information is not recent, then a node is chosen at random.

- The balancer assumes a *completely connected graph* as a logical topology between the nodes. All nodes are within one hop distance of each other.[2]

- A load threshold is used to limit excessive load transfers in the sender mode, and thus avoid *load thrashing* common in sender-initiated balancers.

The four phases in dynamic load balancing are - load evaluation, load balancing profitability determination, task selection, and task migration. The second phase is more common in process-based parallel systems, and is not affordable for fine-grain multithreading due to its high cost. Load transfers are always assumed profitable because the goal here is to minimize idle time rather than balancing load equally. Task selection in the third phase is automatic in the EARTH model, due to the DEQUE structure for the token queue. Tokens from the top of the token queue are always chosen to migrate.

Transfer Policy: This policy determines the balancer initiation strategy, i.e. whether the current load situation warrants the initiation of load transfer.

The receiver mode of the *Rand* balancer is switched on in three situations:

- The scheduler finds no ready thread to execute in both the ready queue and token queue.

- On receiving a load request, two scenarios are possible: a receiving node for the load request is already idle; after responding to the request, a node finds an empty token queue.

[1]The term *hybrid* is known as *symmetric* elsewhere in the literature.
[2]This may not be true in the underlying physical architecture.

- After receiving a token from the network, an idle node notices that it has no extra tokens.

The sender mode of the *Rand* balancer is switched on, when the number of entries in the local token queue equals a particular threshold. The threshold is calculated by an empirical formula, which takes into account the number of nodes in the system. This formula has been fine-tuned experimentally until satisfactory results are observed. The threshold is double the number of probes sent. The number of probes sent, in turn, depends on the number of nodes in the execution. If the threshold is computed to be less than five, it is normalized to value that provides a good balance between aggressive and conservative load migration.

Selection Policy: This policy selects the token for migration. There is no extra effort invested in identifying a suitable token to migrate. The token queue simplifies this policy, as the tokens at the top of the token queue are expected to be higher in the activation tree, are expected to have more work. Locality requirements of communicating threads are taken care of, by adopting a depth-first search pattern for local thread execution.

Location Policy: A partner for load migration is identified in this policy. In the receiver mode, the objective of the balancer is to find the richest possible node in the whole system. Load probes are sent to randomly chosen nodes. Once all the probes are acknowledged with the load status on remote nodes, the richest node is determined by comparing the load status from the probes. Finally, a token request message is sent to the richest node.

An important variable is the number of load probes. The number chosen should be large enough to represent the total number of nodes in the execution, but at the same time must be small enough not to cause an explosion of load probe messages, and as a result unacceptable load overheads. An empirical formula after thorough experimentation has been determined for the number of load probes.

$$d = (Number\ of\ Nodes)/10 + 1)$$

The value for d, the number of load probes is constant for a whole execution. It has worked very well for the portable EARTH runtime system on the IBM SP-2. Its value may need a change on other parallel systems, like the Fast Ethernet based Beowulf system, where the network latencies and polling overheads are relatively high.

In the sender mode, the load state information database on each node is checked for the poorest, or the most idle node in the system. If this cannot be determined from the

database, then a node is chosen at random. A message with a token is sent to the chosen node. No load probes are sent during the sender mode. Making the most accurate choice in the sender mode is not as important as in the receiver mode. While care is taken to make the best choice with available information, there is no need to spend too much CPU time on load balancing overheads, especially when the local processor has enough application threads to execute.

Information Policy: A global load information database is maintained on every node. Initially ambitious plans with sophisticated search patterns were conceived, but our experiments have convinced us of the need for a simple, and effective database access. Instead of checking the load information for every node, a reverse approach is adopted in the design of the data structure for the database. A single-dimensional array indexed by load is maintained. Each of the elements of the array have a structure with fields for a node number, and a reuse flag. As the balancer is always searching for the richest, or poorest nodes, this approach works fine. Whenever load information is received, the node is entered into the array slot with corresponding load index. The reuse flag is set to indicate that this information is very recent. Once the information is used in making a decision, the reuse flag is reset in order to avoid using old and inaccurate load information.

Load information is collected in three ways: load probes, load balancing messages, and piggy-backed information. When a load probe is acknowledged, the data is stored in the database. Also, when a load request or load probe is received, then it is easy to assume that the sender has zero workload. Finally, local load information is piggy-backed over token transfer messages sent abroad.

Whenever a load balancing message is received at a node, a corresponding handler is invoked. The behavior of the *Rand* balancer for different load balancing messages is listed below:

Receiving a load probe : Local load information (entries in the token queue) is composed into a message and sent to the sender node. An entry is made in the load information database, to record the idle state of the sender node.

Receiving probe data : Store the load information in the database. If all the load probes are acknowledged, then compare the data from all the probes to determine the best destination. Then, a token request message is sent to the chosen node.

Receiving a token request : Update the node information in the load database, to reflect the idle status of the sender node. If token queue is not empty, respond with a token

to the request. After the token transfer,if the token queue is empty, then initiate the receiver mode of the load balancer. On the other hand, if there were no tokens when the request was received, initiate the receiver mode of the balancer. Note that, a token request is satisfied, even if it is the only token in the token queue. Previous experience has shown us that, performance degrades if a token request is not satisfied due to the lack of spare tokens.

Receiving a token : Down-load the piggy-backed load information into the database. If this node is idle, consume the token, and send a token request. If it is not idle, and the number of tokens in the token queue equals the load threshold, then initiate the sender mode and forward the token to a chosen destination. If the workload on the current node is below the threshold, add the token to the token queue.

Sending a token request: If the number of load probes for this execution is one, then select a node at random. Otherwise, issue d number of load probes to randomly selected nodes.

3.3 Other Balancers

This section describes the algorithms for eight other balancers implemented in the EARTH runtime system. Six of these balancers are implemented in order to highlight the individual significance of various features that together form the *Rand* balancer. The *Minima* balancer does not perform any load balancing, and therefore provides the lower bound for parallel performance. The *Central* balancer implements a centralized load balancing algorithm on distributed memory machines, and is intended to point out th resulting degradation in performance. The load balancers are as listed below:

Receiver-Initiated : *Rand-Rcv-Info, Rand-Rcv*

Sender-Initiated : *Rand-Snd-Info*

Hybrid : *Rand-Hybrid-Noinfo, Rand-Hybrid, Rand-Hybrid-Piggyback*

Performance Bound : *Minima*

The balancers based on the randomizing algorithm allow a comparative study of the *Rand* balancer with respect to their transfer policy, and information policy. The studies

on the transfer policy establish the fact that the hybrid nature of the *Rand* balancer outperforms the sender/receiver-initiated versions of itself. Comparison of different levels of sophistication in the information policy confirms the soundness of the information policy in the *Rand* balancer.

3.3.1 Receiver-Initiated Balancers

The *Rand-Rcv-Info* balancer is similar to the *Rand* balancer with its sender component excluded. This balancer is strictly receiver-initiated. Whenever, there is a shortage of tokens, a token request is sent abroad. Token transfers occur only as response to token requests. However, the selection, location, and information policies remain the same as the *Rand* balancer. A database is maintained on each node with load information collected from probe data, load messages, and piggy-backed data. The load database is used in choosing a destination for load request, when there is only one ready thread in the ready queue. This balancer compares the receiver-initiated policy versus the hybrid policy of the *Rand* balancer.

The *Rand-Rcv* balancer differs from the *Rand-Rcv-Info* balancer in that, no load information is considered in choosing the destinations for load requests. This balancer is to highlight two issues in receiver-initiated balancers: significance of the super-market model, where load probes are sent to randomly chosen nodes, and the node with highest workload is chosen as the target for token request; and the relevance of information policy in token transfers. This balancer uses a simple work-stealing algorithm in which the destination node is picked on random. No load probes are sent, nor any load information database is maintained.

3.3.2 Sender-Initiated Balancers

The *Rand-Snd-Info* balancer is a sender-initiated balancer. It differs from the *Rand* balancer in two respects: there is no receiver-initiated component; and no load probes are sent. Barring these two differences, the balancer is similar to the *Rand* balancer.

Load probes are not used in order to avoid instability in the system due to very high balancer related message traffic. However, a load information database is maintained on each node. Load information gleaned from load messages, and piggy-backed messages is mapped into the database.

A load threshold is used to initiate load transfers. When workload on a node equals

the threshold, a target node is chosen either from the current load information, or by using a randomizing function.

3.3.3 Hybrid Load Balancers

The *Rand-Hybrid-Noinfo* balancer differs from the *Rand* balancer in that it does not maintain a load state database on each node. The *Rand* balancer uses global load state information to a small extent in the receiver mode, and in a major way in the sender component. This use of load state information is excluded in the *Rand-hybrid-NoInfo* algorithm. In the receiver mode, target nodes are chosen by sending random probes, and selecting the most appropriate node. In the sender mode, a load destination is picked at random. The objective of this balancer is to highlight the difference that the load information policy in the *Rand* balancer makes to total elapsed time.

The *Rand-Hybrid* balancer is an extension to the *Rand-Rcv* balancer in that, a sender component also is also included. This balancer is a simple hybrid balancer in which target nodes for both token requests, and token transfers are chosen at random. The major difference with the *Rand* balancer lies in the location policy and information policy. No load probes are sent, neither is any global load state information considered in the decision making process. this balancer is created to observe the performance of a naive, hybrid, randomizing algorithm. Its performance makes the benefits of load probes, and information policy in the *Rand* balancer obvious.

The *Rand-Hybrid-Piggyback* balancer extends the *Rand-Hybrid* balancer by considering load state information collected from load messages and piggy-backed messages, before choosing a destination node for token transfer. The performance of this balancer, when compared to the *Rand* balancer, isolates the effect of load probes and accurate global load information on application performance.

3.3.4 Performance Bound

The *Minima* load balancer provides a realistic lower bound for performance in the "parallel" execution of Threaded-C programs. It is basically a dummy load balancer. It does not do any load balancing.

The *Minima* balancer is created as an improvement over the *Nop* balancer [33] in the EARTH runtime system. The *Nop* balancer launches the executable[3] on all the nodes in

[3]Combination of application code and runtime system code

the execution, with one process on each node. After this, unless the programmer maps some work onto a node, the whole work is executed on node 0. Runtime load balancing is non-existent. However, this policy does not provide a realistic lower bound of parallel performance. Unless application threads are running on all the nodes, it cannot be termed a parallel execution. What the *Nop* balancer provides is a sequential execution, combined with the overheads of maintaining a parallel environment.

The *Minima* balancer avoids this scenario by ensuring that every node gets to execute at least one token. Of course, this is subject to the availability of parallelism in the application. Once each node receives one token, the runtime load balancer is switched off. Parallel execution proceeds without any more load balancing, and the total elapsed time reflects the lower bound in parallel performance. The performance of the *Minima* balancer can be compared to other balancers to determine their improvements/overheads. In other words, the *Minima* balancer provides a realistic, experimental lower bound for parallel application performance.

In order to initiate parallel execution of the activation tree, every rich node passes its second token to its logical neighbor, i.e. node 0 passes a token to node 1, node 1 to node 2, and so on, until the final node - $NUM\_NODES - 1$ is reached. A node is *rich* if it has at least one token. After exporting a token abroad, each node consumes all the tokens it generates. There is no load balancing through the remaining part of the application execution. In addition to the token and its children that a node gets to compute, a node may also receive tokens when the programmer/compiler map certain workload onto a node using the `INVOKE` instruction.

Chapter 4

Experimental Framework

In this chapter, we present the framework in which the different load balancers in the portable EARTH runtime system on the IBM SP-2 system are evaluated. Initially, we enumerate the benchmarks used to study the suitability of various balancers to different applications. Second, we briefly review the experiments planned, and particular characteristics of the load balancers which are under observation. Finally, we describe the hardware platform on which these experiments are performed.

4.1 Benchmarks

The domain of applications considered in this study are Threaded-C programs belonging to the divide-and-conquer, regular, and irregular classes of applications. These applications are characterized by fine-grain threads with very short run-times, frequent communications and synchronizations, and varying amounts of parallelism that can be exploited by the runtime system. Therefore responsiveness, ability of the balancer to choose the right destination (either for a sender or receiver) in minimum steps, and minimum balancer overheads are crucial for better performance [34].

The benchmark programs used in our experiments are taken from the EARTH Benchmark Suite (EBS) [160]. Table 4.1 gives a brief overview of these benchmarks. *Fibonacci*, *N-Queen* and *TSP* (Traveling Salesperson Problem) are typical examples of recursive divide-and-conquer algorithms. The *Paraffins* benchmark is somewhat special in that it generates very irregular load units and has only a short execution time. *Matrix Multiply* and *Tomcatv*, on the other hand, perform regular SPMD computations.

Benchmark Name	Problem Domain	Type	Tokens Generated	Threads executed
Fibonacci (33)	Combinatorial	Divide and conquer	11405772	17108661
N-Queen (12)	Graph Searching	Divide and conquer	9916	24791
TSP (10)	Graph searching	Divide and conquer	5861	18407
Knary (7,7,2)	Computation Trees	Divide and Conquer	98040	274516
Matrix Multiply	Numerical Computation	Regular SPMD	NA	NA
Tomcatv (257)	Scientific Computation	Regular SPMD	101	304
SPMD (4,4,0)	Scientific Computation	Regular SPMD	2100	4301
Paraffins (28)	Chemistry	Irregular	1843	1904

Table 4.1: The EARTH Benchmark Suite

The Fibonacci benchmark is programmed in a recursive fashion, as per the divide-and-conquer programming model. Each token does very little work other than spawning two children. The Fibonacci program presents an interesting problem - how to tackle extremely fine-grain applications. Secondly, this program also showcases the ability of the multithreaded environment to create, maintain, and terminate a large number of threads with minimum overheads. As shown in Table 4.1, the Fibonacci(33) problem creates 11405772 tokens, and 17108661 threads. This benchmark represents a challenge to the load balancer, not because of any difficulties in understanding the program behavior, but due to its very fine-grain threads. In order to achieve any kind of improvement, the load balancer has to be very simple, with absolutely minimum overheads, and perform load balancing only when required.

The N-Queens is a typical recursive program that counts how many ways N queens can be placed in an $N \times N$ chess board so that no queen may attack another. In the version that we used, $N = 12$, and the parallelism is "throttled". When four queens are placed on the board, the program switches to a sequential execution and no longer generates migratable tokens. The idea is that at the level of the recursion enough instantiations of the recursive function have been generated to distribute the computation among the processors in the machine. Our implementation, initially expands the board with breadth-first search, and then switches to depth-first search. In order to coarsen the grain size, a throttling threshold is used.

The Traveling Salesperson Problem (TSP) is another graph-theoretic problem. Here, the aim is to find a Hamiltonian tour when a traveling salesperson visits N cities, each city exactly once, and returns to the city of origin. The salesperson is expected to cover

all cities and optimize cost for the whole trip. A complete weighted graph is used to represent the cities, and the costs of inter-city travel.

The *K-nary* models the divide-and-conquer strategy. It generates a k-ary computation tree, i.e. each node has k children [34]. By changing the depth and width of the tree, we can simulate many common situations. The *Knary (n,k,r)* represents a k-ary tree with depth n and r children being executed locally. Some knary trees have special interest to load balancing studies. For instance, Knary (2, 512, 0) is a two-level knary tree. The root of the knary tree generates 512 children and waits for their termination. These 512 children are in the form of tokens, and are free to relocate to remote nodes. As these tokens are created on one node initially, the speed with which they are distributed determines the elapsed time.

The dense matrix multiply algorithm that we used in this study is a simple minded, non-blocking algorithm that computes $C = A \times B$, where A, B and C are $N \times N$ matrices (in our measurements $N = 1024$). Both matrices A and B are stored in node zero and the resulting matrix C is to be also stored in the memory of node zero. Node zero generate migratable tokens that are to compute one row of the matrix C and move the result back to node zero. The first time that a node i executes a token, it copies the entire matrix B to its local memory and the specified row of A. It retains the copy of B to reuse in the computation of future tokens. Although a dense matrix multiply is a very regular algorithm, this version relies in the dynamic load balancer to distributed the load among the processors.

The Tomcatv is a floating point SPEC92 benchmark and represents large data-parallel applications [33] with 257 $\times$ 257 meshes. initially, each iteration updates the meshes using near-neighbor calculations, and then by performing calculations with horizontal loop-carry dependencies. Separate rows synchronize with each other using a pure data-flow paradigm. The data set is fixed to a node, while tokens migrate between the nodes. The token migration is decided by the dynamic load balancer, rather than by compile-time partitioning of the problem. From a modeling point of view, this applications highlights the provision in Threaded-C to perform peer-level synchronizations between nodes at the same level in an activation graph.

In the SPMD model, loop indices are divided among the nodes statically. This can lead to poor performance when the execution time is not the same for all indices. *Dynamic SPMD*, on the other hand, is a more flexible approach that relies on the load balancer to distribute the parallel loops [34]. This results in a large number of tokens being

generated for each loop, after which node 0 waits for all iterations to complete, performs some sequential computations, and then starts another parallel loop. The generalized *dynamic spmd* program computes a *Knary (n,k,r)* tree in each one of these iterations. This application models a typical barrier-synchronized application. In order to achieve good performance on such programs it is important to minimize the time needed to distribute the tokens to all nodes and the time to achieve an even load distribution.

Paraffins is one of the four "Salish-an problems" from the 1988 Salish-an High-Speed Computing Conference. Paraffins enumerates all distinct isomers of each paraffin (molecule of the form C_nH_{2n+2}) of size up to a given maximum. The problem solved by paraffins is similar to the problem of detecting isomorphisms in labeled free trees. A list of paraffins is generated and the program returns an array filled with the number of distinct paraffins of each size up to and including the maximum. To exploit parallelism, functions are invoked in all the processors to compute the radicals and then tokens are generated to compute the paraffins of the required size. This benchmark belongs to the irregular class of problems, with irregular communication patterns, and unbalanced computations. In our experiments we measured the performance for Paraffins(28).

4.2 Performance Evaluation

We identify different parameters that influence program performance, and study their effect with respect to different load balancers. The objective is to identify ideal load balancer policies for different application load situations, and arrive at a lowest denominator balancer that performs relatively better in most situations.

We have implemented ten dynamic load balancers, and compare their performance against seven existing balancers. Initially, we compare distributed dynamic load balancing against centralized dynamic load balancing for distributed memory machines. Then we study the benefits of a randomizing load balancer in a fine-grain multithreading environment with varying application and workload parameters, and compare its performance against seven existing balancers. We identify the different factors that have contributed to the relatively better performance of the randomizing algorithm, by comparing it against different versions of itself, each with varying degrees of sophistication. Finally, we review the advantages of different dynamic load balancer policies against a situation where there is no load balancing.

The performance at varying workloads with different benchmarks is observed for each

load balancer. In each of the cases, the elapsed time, idle time, number of balancing activities, token distribution, percentage of migrated tokens, etc. are measured. The time spent on different runtime system activities is documented. In addition, changes in performance are noted with varying architectural parameters like polling interval, number of nodes, communication topology, token prefetching, and application parameters like workload, grain size, call-graph size and shape.

Finally we measure the latencies and overheads associated with EARTH operations, and make a comparative study of EARTH operations on three different implementations of EARTH.

4.3 EARTH-SP Implementation

The EARTH-SP system realizes the EARTH model on the IBM SP-2 system. The IBM RS/6000 Scalable POWER Parallel System (SP-2) is a distributed memory multiprocessor [8]. Each processing node is equipped with a 120 MHz POWER2 Super Chip, 128 KB of data cache, 32 KB of instruction cache, at least 64 MB of RAM, and operate with a 256 bit memory bus. The tb-3 switch provides a network interface with a peak hardware bandwidth of 150 MB/s in each direction. A detailed description of the EARTH-SP2 implementation is provided in [92, 33].

The POWER2 Super Chip is an improvement of the POWER2 processor. Its main features include: dual floating point and fixed point units, peak execution rate of 6 instructions per cycle, improved instruction set (quad-word load/store, zero-cycle branches, hardware square root, etc.)

The SP2 high performance switch is a connecting network which allows any node on the SP2 to communicate directly with any other SP2 node [37]. The switch is a high bandwidth, low latency, bidirectional, multi-stage, omega, buffered-wormhole routing packet switch [88, 87]. The tb-3 switch interfaces between the network switch and the compute node. The tb-3 card data structures are mapped into the user space, and can be accessed from the application program. In our case, the application interface that accesses the tb-3 card data structures is the communication layer in the portable EARTH runtime system. The performance benefits by accessing the send/receive buffers in the network switch interface far outweigh the advantages of using the communication layer of the MPI provided on the IBM SP-2. Modularity of the communication layer in the runtime system ensures that the rest of the runtime system code is independent of the tb-3 switch

interface, and therefore is portable.

User jobs are submitted in batch mode using the EASY-LL batch system on the IBM SP-2 [38]. Load Leveler is a batch system originally developed for the IBM SP-2, and it allocates resources across a network while attempting to maintain a balanced load, fair scheduling, and an optimal use of resources. EASY was originally developed at the Argonne National. The EASY algorithm schedules the job queue on a FCFS basis. It allows smaller jobs further down the queue to run as long as they complete before the waiting job ahead in the queue is scheduled to run. The EASY-LL is a collaboration of the EASY and LoadLeveler algorithms.

Chapter 5

Performance Results

In this chapter, the performance results of the load balancers presented earlier for applications described in chapter 4 are studied, and the behavior of the balancers for different work descriptions is analyzed.

The main results of this study are listed below:

- For irregular and highly recursive programs, it is beneficial to generate large (abundant) number of threads to facilitate the work of the load balancer. See section 5.1.
  - Furthermore, a randomizing algorithm (*Rand*) gives the best performance as long as the cost of computing the random number does not dominate the overall time of thread execution.
  - When it is not favorable for applying the *Rand* balancer, a hybrid history information based algorithm (*His*), a simple work-stealing algorithm (*Spn*) are preferable in the descending order.
- The *Rand* balancer is *good* for fine-grain applications. An in-depth study of the *Rand* balancer is performed. See section 5.2.
- In order to understand the various factors that contribute to the good performance of the *Rand* balancer, a comparative study of the *Rand* balancer, and different versions of itself each with varying degrees of sophistication, is performed. See section 5.3.
- When the *Rand* balancer does not perform well, a detailed study is performed on alternate balancers. See section 5.4.

- A spectrum of experiments are designed to understand application behavior with different load balancers. See section 5.5.

- An analysis of the overheads and latencies of various multithreaded operations supported in the EARTH system, shows that it is possible to emulate a multithreaded environment in software with minimum overheads, and derive scalable performance for fine-grain applications. See chapter 6.

- The ratio of CPU speed to network speed is a crucial factor that determines performance of EARTH applications across a range of machines. Besides network bandwidth, costs associated with the network interface in the runtime system also makes a significant impact on application performance. See chapter 7.

Each of the above points are discussed in detail in the following sections.

5.1 Overall Performance

For irregular and highly recursive programs, it is beneficial to generate large (abundant) number of threads to facilitate the work of the load balancer.

- Furthermore, a randomizing algorithm (*Rand*) gives the best performance as long as the cost of computing the random number does not dominate the overall time of thread execution.

- When it is not favorable for applying the *Rand* balancer, a hybrid history information based algorithm (*His*), a simple work-stealing algorithm (*Spn*) are preferable in the descending order.

There are two minimum conditions for load balancing to be successful. Firstly, there should be enough parallelism to exploit in the application. Application parallelism depends on the programming model of the application, and on the input workload. While the programming model is a characteristic of the program, the input workload is a property of a particular execution of the application. The input workload determines the number of available threads in a particular execution, and as a result the number of ready threads at any point of time during the execution. A split-phase nature of threading models, as in Threaded-C, allows to start executing another ready thread while the current thread

meets a long-latency operation. There is higher probability of a successful choice of a destination in load balancing, if the number of ready threads in the execution is high.

Secondly, the work migrated should dominate the load balancing overheads. As the grain size of work decreases from the whole program (in sequential execution) to a set of instructions (threads), the amount of parallelism in the application increases. But at the same time, the load balancer has to become more lean and inexpensive in order to be profitable. An illustration of this feature can be observed in Figs. 5.3, 5.36, 5.27, 5.28. Average token size in the Fibonacci is around 2 μ s on the IBM SP-2. As the input workload increases, however, the amount of parallelism increases, and also the amount of work transfered is considerably higher than typical balancer overheads. This is possible due to the token queue in the EARTH runtime system which allows work at higher levels in the activation tree to be migrated to remote nodes.

Benchmark	Dual	Spn	Shis	Snd	His	Range	Catapult	Rand
Fibonacci(33)	1.14	1.14	13.66	OF	1.19	1.21	1.2	1.02
Queens(12)	0.24	0.167	4.71	0.171	0.176	0.175	OF	0.165
TSP(10)	0.43	0.32	7.8	0.36	0.28	0.29	0.28	0.27
Knary(7, 7,2)	2.13	0.93	24.76	1.037	0.908	0.94	0.95	0.906
Knary(2,512,0)	0.054	0.013	0.169	0.085	0.0076	0.014	OF	0.015
Matrix(1024X1024)	70.31	49.53	293.79	17.52	12.21	14.66	63.42	16.96
Tomcatv(257)	2.45	1.78	OF	OF	0.54	0.39	OF	5.6
SPMD(1,1,0)	0.25	0.16	0.68	0.08	0.11	0.1	0.63	0.15
SPMD(4,4,0)	1.9	0.72	14	0.63	0.86	1.27	13	0.79
Paraffins(28)	7.43	6.55	104	7.54	6.54	6.79	OF	6.46

Table 5.1: Overview of Results. Elapsed times in **seconds** are shown for different benchmarks belonging to the recursive (divide-and-conquer), regular and irregular programming models against various dynamic load balancers belonging to the receiver-initiated, sender-initiated and hybrid categories. Measurements are based on **32 node** runs. These elapsed times include the time spent on profiling the runtime system actions. Table E.1 shows elapsed times without profiling effects.

Table 5.1 shows the elapsed times for different balancers for applications belonging to a wide-ranging set of programming models, and load situations. The divide-and-conquer, regular, and irregular classes of applications are considered for experimental evaluation of the balancers. In addition, applications with very low workloads are included to study the ability of the balancer to distribute load rapidly, and also to showcase the potential of a balancer for very low load situations where the emphasis is on minimum balancer

overheads rather than maximum processor utilization.

The relative ranking of the balancers for different applications is shown in Table 5.2. The *Rand* balancer performs very well for divide-and-conquer, and irregular classes of applications. This can be attributed to the ability of the *Rand* balancer to distribute load equally among all nodes in the system, which in turn is a result of the accuracy of its load balancing decisions. Another reason is the relatively better scaling of both thread execution time and overheads in the *Rand* balancer. While it is good for regular applications, the *His* balancer is more preferable. Gathering load information is an unnecessary overhead for regular applications, where the workload is more or less evenly distributed. A history information based balancer is more than adequate to address the minor load imbalances.

Benchmark	Dual	Spn	Shis	Snd	His	Range	Catapult	Rand
Fibonacci(33)	3	2	7	8	4	6	5	1
Queens(12)	6	2	7	3	5	4	8	1
TSP(10)	7	5	8	6	3	4	2	1
Knary(7, 7,2)	7	3	8	6	2	4	5	1
Knary(2,512,0)	6	3	7	2	1	4	8	5
Matrix(1024X1024)	7	5	8	4	1	2	6	3
Tomcatv(257)	4	3	8	8	2	1	8	5
SPMD(1,1,0)	6	5	8	1	3	2	7	4
SPMD(4,4,0)	6	2	8	1	4	5	7	3
Paraffins(28)	5	3	7	6	2	4	8	1
Average	5.5	3.3	7.6	4.5	2.7	3.6	6.4	2.5
Rank	6	3	8	5	2	4	7	1

Table 5.2: Relative ranking of the different balancers based on their performance as shown in Table 5.1

For barrier-synchronized applications (SPMD), the *Snd* balancer is a clear winner. Barrier-synchronized applications place two challenging demands on a load balancer: first, they are traditionally low-load applications, i.e. irrespective of the input workload, the number of tokens in every phase cannot be substantially higher than the number of nodes; secondly, the fast token distribution capability of the load balancer is of prime importance. Usually, a node 0 computes work for the next phase, and issues it, and when these tokens are consumed by other nodes and synchronization has taken place among all the nodes, the node 0 issues the next set of tokens. The *Snd* balancer performs well

here due to its ability for rapid disposal of tokens to other nodes in the system without spending too much time in the target deciding phase. As the negative impact of instability is only possible for high load situations, the sender-initiated balancer does very well for barrier-synchronized applications. If the workload for a barrier-synchronized application is reasonably higher, the *Rand* balancer performs at a respectable third position.

For very low load applications (Knary(2,512,0), SPMD(1,1,0)), the balancers which perform effective load distribution at minimum overheads do well. The *Rand* balancer understandably cannot win this race, due to its relatively longer decision making phase.

On the whole, the *Rand* balancer performs the best on a wide range of applications. Another hybrid balancer, but based on history information rather than global load information (*His*), comes a clear second. A simple receiver-initiating balancer, the *Spn* balancer, results in agreeable performance for fine-grain applications due to its low overheads, and comes third.

The *Range* balancer is the fourth best balancer. This confirms the limited use of the range list information. The sender component of the *Range* balancer sends extra tokens to the far node in the range list. It works reasonably well for low load situations, because the far node is more likely to be idle than the near node as load state fluctuates very rapidly in low load situations. However, for high load situations, the impact of the range information is less significant. because As shown in Table 5.2, the *Range* balancer does well only in low load applications.

The *Catapult* balancer is one of the poor performers. Despite its similarities with the *Range* balancer, the *Catapult* balancer ranks well below the *Range* balancer. Furthermore, in many instances, it causes an explosion of balancer related message traffic and terminates the application. The reasons are not difficult to observe. A *Range* balancer differs from the *Catapult* balancer in two ways:

- when a node receives a token (if it is wealthy), the *Catapult* balancer passes the token to the nearest node in it's range list, while the *Range* balancer passes it to it's predecessor.

- The *Range* balancer after passing the token to it's predecessor,sends an update message to the sender of the token (asking to be removed from its range list).

The ring topology assumed in the *Dual* balancer severely limits its scalability, ability to respond rapidly to fluctuating load situations, and fast token distribution capabilities. However,its algorithm is extremely simple, that makes it very useful for applications with

very fine-grain threads, and low load situations. The *Shis* balancer causes high message traffic, and indicates the non-utility of the history information for purely receiver-initiated balancers.

Benchmark	Time (secs)	Dual	Spn	Snd	His	Rand
Fibonacci(33)	Elapsed Time	1.137067	1.136061	OF	1.18523	1.01747
	Execution Time	0.554408	0.550621	—	0.55286	0.442716
	Balancer Overhead	0.12063	0.112915	—	0.141178	0.000359
	Polling Overhead	0.258839	0.262019	—	0.287827	0.247186
	Idle Time	0.041404	0.031044	—	0.052133	0.020562
	Rank	3	2	5	4	1
Queens(12)	Elapsed Time	0.240889	0.167472	0.170896	0.175538	0.165809
	Execution Time	0.156755	0.159375	0.164493	0.161955	0.155494
	Balancer Overhead	0.001435	0.001458	0.002793	0.00534	0.000682
	Polling Overhead	0.033116	0.018975	0.020644	0.020861	0.017798
	Idle Time	0.095079	0.019033	0.017112	0.022409	0.02154
	Rank	5	2	3	4	1
Knary(7,7,2)	Elapsed Time	2.126129	0.926829	1.03691	0.907591	0.906191
	Execution Time	0.904189	0.899871	0.952327	0.893569	0.893569
	Balancer Overhead	0.008455	0.007272	0.033319	0.002861	0.001547
	Polling Overhead	0.249025	0.021644	0.051247	0.016732	0.015995
	Idle Time	0.913451	0.024985	0.081386	0.013053	0.012389
	Rank	5	3	4	2	1
SPMD(4,4,0)	Elapsed Time	1.903876	0.719708	0.632134	0.859295	0.794468
	Execution Time	0.454005	0.474271	0.469349	0.550345	0.462295
	Balancer Overhead	0.008217	0.0479	0.02079	0.139989	0.017921
	Polling Overhead	0.285509	0.074199	0.050056	0.115658	0.078894
	Idle Time	1.255786	0.170489	0.164918	0.114992	0.326268
	Rank	5	2	1	4	3
Paraffins(28)	Elapsed Time	7.42791	6.554465	7.541764	6.543834	6.458399
	Execution Time	6.564703	6.412043	6.570584	6.390272	6.393398
	Balancer Overhead	0.002117	0.002517	0.001366	0.001039	0.001359
	Polling Overhead	0.232714	0.096929	0.251115	0.0962	0.079782
	Idle Time	0.875255	0.155766	0.983546	0.165938	0.077731
	Rank	4	3	5	2	1

Table 5.3: A breakup of the total elapsed time. Execution time corresponds to the time spent on executing application threads. All time measurements are in **seconds**. Measurements are based on **32 node** runs. All measurements except the elapsed time are average of values from 32 nodes.

The average values on 32 nodes for thread execution time, and other overheads inherent in a parallel environment are shown in Table 5.3. A breakup of the total elapsed times,

corresponding to those in Table 5.1. The *Rand* balancer provides a better balancing of work and overheads on all the nodes in the execution. For the divide-and-conquer classes of applications (Fibonacci, Queens, and Knary), the application thread execution time for the *Rand* balancer is less than or equal to that of other balancers. In those cases where thread execution time is equal to that of other balancers (Knary), the *Rand* balancer scores better due to its relatively low overheads and idle time.

For barrier-synchronized applications like the SPMD, the *Rand* balancer spends more time in polling overheads when compared to the *Spn* and *Snd* balancers. Due to the low load situation, the receiver component of the *Rand* balancer sends lots of load probes leading to increased network communications. This also increases the per-node idle time.

For irregular applications like the Paraffins, despite having a slightly higher per-node thread execution time than the *His* balancer, the hybrid nature of the *Rand* balancer responds very well to the irregular parallelism and minimizes idle time significantly.

The profile data collected during an 8-node execution is shown in Table 5.4. A list of the profile data collected at runtime is summarized in chapter C. The data shown here is the number of acts under each category. Remote communications are the total number of messages sent abroad in order to satisfy global memory and synchronization requirements. Balancing acts is the sum of requests sent, requests received, tokens sent, tokens received. Idle time is the percentage of balancer related idle time in the total elapsed time.

Normally 8 nodes is a small number for the *Rand* balancer to scale well when compared to other balancers. However, the randomizing algorithm starts giving early gains for irregular applications, such as the Paraffins.

5.2 Rand Balancer

The *Rand* balancer is *good* for fine-grain applications. An in-depth study of the *Rand* balancer is performed.

- The hybrid (symmetric) nature of the *Rand* balancer, its use of load state information, and the completely connected graph topology assumed between the nodes, are the most important factors for its good performance.
- Accurate global load state information is a crucial component of the *Rand* balancer, and an important reason for its good performance. This result is in contrast to

App-Balr.	Activity	Nodes							
		0	1	2	3	4	5	6	7
	Toks. Gen.	1454	1218	1100	1104	1286	1276	1306	1172
	Toks. Con.	1423	1244	1156	1125	1249	1277	1284	1158
His	Threads Run	3574	3097	2862	2802	3141	3192	3221	2902
0.6384s	Rem. Comms.	118	157	160	96	172	119	100	149
	Bal. Acts	518	559	491	392	626	427	438	651
Queens	Extra Tokens	71	97	84	61	72	58	61	118
(12)	Idle Time	0.17	0.81	0.81	0.57	0.59	0.41	0.34	0.38
	Toks. Gen.	1140	1210	1280	1108	1174	1328	1192	1484
	Toks. Con.	1128	1206	1276	1098	1185	1351	1186	1486
Rand	Threads Run	2827	3017	3192	2750	2957	3366	2968	3714
0.6404s	Rem. Comms.	117	73	64	61	49	61	60	79
	Bal. Acts	2322	2054	1246	1057	1087	1520	1209	1238
	Extra Tokens	802	741	368	365	395	443	403	391
	Idle Time	0.16	0.89	0.75	0.67	0.62	0.55	0.33	0.27
	Toks. Gen.	2772	2000	2032	2020	2008	2012	2024	1932
	Toks. Con.	2096	2090	2106	2104	2113	2114	2084	2093
His	Threads Run	4293	4180	4212	4208	4226	4228	4168	4186
0.5372s	Rem. Comms.	694	774	769	767	768	806	805	853
	Bal. Acts	14999	14541	14341	14331	13901	14265	14332	14308
SPMD	Extra Tokens	2089	2273	2203	2219	2131	2251	2260	2266
(4,4,0)	Idle Time	5.8	9.9	9.2	9	9	8.9	9.6	9.2
	Toks. Gen.	2940	2052	1948	2004	1956	1952	1952	1996
	Toks. Con.	2137	2018	2109	2117	2118	2088	2089	2124
Rand	Threads Run	4375	4036	4218	4234	4236	4176	4178	4248
0.5637s	Rem. Comms.	539	524	505	459	501	529	499	504
	Bal. Acts	15501	11497	8170	7974	8119	8962	8277	8245
	Extra Tokens	2097	1407	1045	982	992	1111	1008	1025
	Idle Time	6.88	16.85	16.35	16.95	16.39	16.95	17.39	16.02
	Toks. Gen.	254	302	319	212	178	230	278	266
	Toks. Con.	185	250	354	263	182	246	284	275
His	Threads Run	202	265	373	277	194	263	298	285
26.47s	Rem. Comms.	9	64	81	69	42	26	20	21
	Bal. Acts	1814	3280	592	888	797	2429	91	91
Paraffins	Extra Tokens	414	746	145	212	192	610	20	20
(28)	Idle Time	0.01	0.09	0	0.03	0	0	0.01	0
	Toks. Gen.	253	232	355	252	231	208	267	241
	Toks. Con.	181	144	528	262	243	197	277	207
Rand	Threads Run	198	154	542	274	263	214	291	221
26.398 s	Rem. Comms.	9	6	179	46	38	28	30	10
	Bal. Acts	29316	31051	14240	66898	10350	15278	12749	18035
	Extra Tokens	7387	7520	3851	3681	3754	4036	3759	3609
	Idle Time	0	0	0.02	0	0.01	0	0	0

Table 5.4: Node-wise Profiling Data on 8 nodes. The time is in **seconds**.

the common intuition that load balancer overheads spent in the accumulation of global load information dominate any possible performance benefits, and therefore such an information policy is not viable, especially for fine-grain parallelism. The *Rand* balancer overcomes this bottleneck by locating the load information gathering actions in the receiver part of the load balancer, and using this information in the sender part. The information policy is *demand-driven* and *receiver-initiated.*

5.2.1 *Rand* versus *Minima*

- The *Rand* balancer provides the best relative performance against a *no-load balancing* situation for parallel applications.

  The *Minima* balancer ensures that the application is executed in a true parallel fashion, i.e. each node gets to execute atleast one token. After that the load balancer on each node is switched off. This allows us to compare the utility of load balancers against a situation where there is no load balancing in a parallel execution. The performance of the *Minima* balancer can be seen as a higher bound for total elapsed time for parallel applications (or lower bound of parallel performance). Any balancer is expected to do much better than the *Minima* baalncer, and their relative performance against the *Minima* balancer can be used to rank them.

  Table 5.5 compares the performance of current receiver-initiated, sender-initiated, and hybrid balancers in the EARTH runtime system. Also, the performance numbers for the *Nop* balancer are shown. The *Nop* balancer does not perform any load balancing. It differs from the *Minima* balancer in that, it offers basically sequential execution in a multithreaded environment. Unless the programmer specifically launches a token on some node, all nodes except node 0 are idle. Further, the sequential elapsed times are burdened by multithreading overheads. In contrast, the *Minima* balancer provides an upper bound on the total elapsed time for realistic parallel executions.

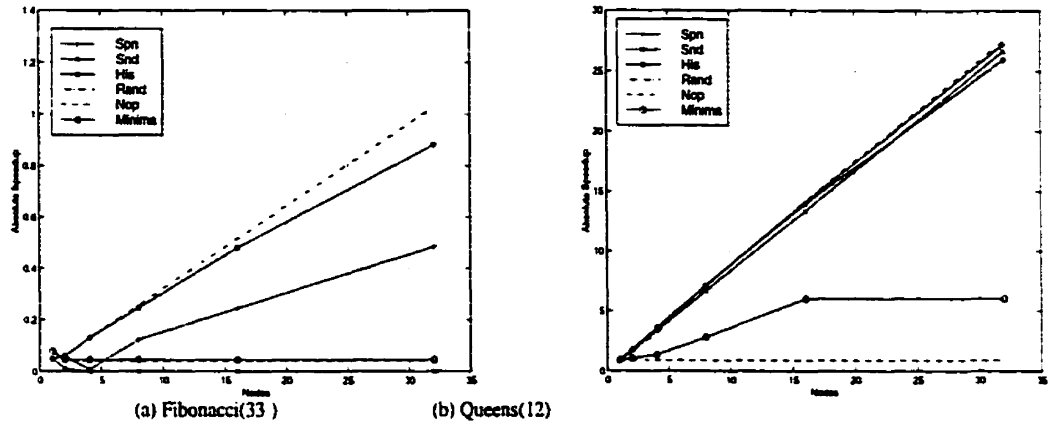
  In Table 5.5, the *Rand* balancer achieves higher speedup and percentage reduction in total elapsed times for the divide-and-conquer and irregular classes of applications. Understandably, it does not do well for barrier-synchronized applications.

  Figs. 5.1, 5.2 show the speedups for different classes of applications for the *Minima* and other balancers. It can be observed that the *Minima* balancer performs distinctly

Benchmark	Attribute	Spn	Snd	His	Rand	Nop	Minima
Fibonacci(33)	Elapsed Time	1.14	OF	1.19	1.02	24.9	23.29
	% Reduction	95.12	–	94.91	95.63	-6.94	0
	Speedup	20.50	–	19.65	22.89	0.94	1
Queens(12)	Elapsed Time	0.167	0.171	0.176	0.166	5.05	0.754
	% Reduction	77.79	77.34	76.72	78.01	-570.34	0
	Speedup	4.50	4.41	4.30	4.55	0.15	1
TSP(10)	Elapsed Time	0.32	0.36	0.275	0.269	8.6	7.78
	% Reduction	95.91	95.34	96.47	96.54	-10.53	0
	Speedup	24.45	21.45	28.35	28.90	0.90	1
Knary(7,7,2)	Elapsed Time	0.93	1.04	0.907	0.906	28.87	24.77
	% Reduction	96.26	95.81	96.34	96.34	-16.57	0
	Speedup	26.72	23.88	27.30	27.33	0.86	1
Knary(2,512,0)	Elapsed Time	0.013	0.0082	0.0073	0.012	0.1684	0.1682
	% Reduction	92.21	95.14	95.67	91.52	-0.10	0
	Speedup	12.84	20.58	23.08	11.80	1.00	1
SPMD(4,4,0)	Elapsed Time	0.72	0.63	0.86	0.79	14.038	14.037
	% Reduction	94.87	95.50	93.88	94.34	-0.01	0
	Speedup	19.50	22.21	16.34	17.67	1.00	1
SPMD(1,1,0)	Elapsed Time	0.161	0.081	0.11	0.15	0.75	0.67
	% Reduction	76.14	87.96	83.19	77.60	-10.61	0
	Speedup	4.19	8.31	5.95	4.46	0.90	1
Paraffins(28)	Elapsed Time	6.55	7.54	6.54	6.46	121.7	118.8
	% Reduction	94.48	93.65	94.49	94.56	-2.41	0
	Speedup	18.13	15.76	18.16	18.40	0.98	1

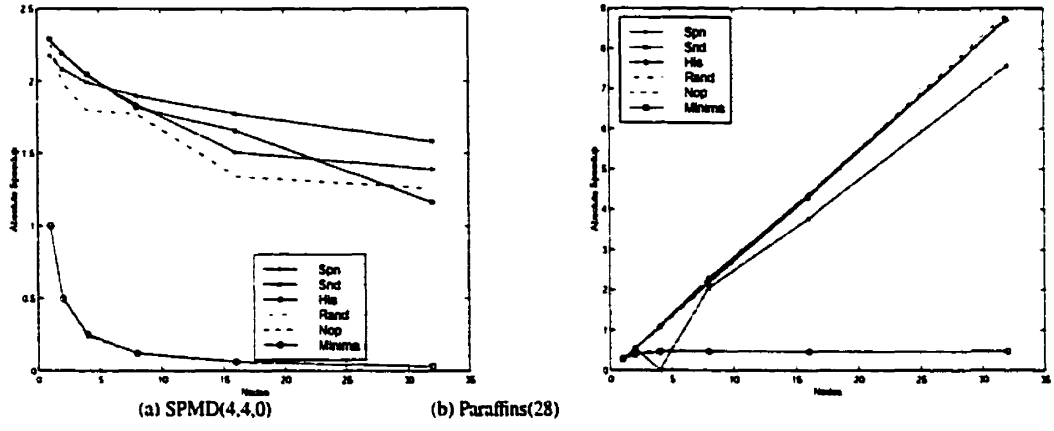
Table 5.5: Performance comparison for all the benchmarks with different load balancer policies. For each benchmark, the first row shows the elapsed times for **32 nodes**, and the measurements are in **seconds**. The second row shows percentage reduction in total elapsed times when compared to the *Minima* balancer. The third row shows speedup for each balancer as compared to the *Minima* balancer.

better than the *Nop* balancer for most of the applications, due to the initial load balancing when each node receives a single token. After this the curve flattens out. Other balancers perform very well when compared to the *Minima* balancer, as expected. Their performance relative to that of the *Minima* balancer isolates and quantizes the peformance benefits from load balancing.



(a) Fibonacci(33) (b) Queens(12)

Figure 5.1: Performance comparison between Minima, Nop and other Balancers of different balancers



(a) SPMD(4,4,0) (b) Paraffins(28)

Figure 5.2: Performance comparison between Minima, Nop and other Balancers of different balancers

5.2.2 Scalability of *Rand* Balancer

- The *Rand* balancer is highly scalable and robust (stable) for irregular and divide-and-conquer (recursive) classes of applications. However, this balancer is not appropriate for regular applications.

  The performance results in this section show the absolute and relative speedups for different applications for different balancers. Absolute speedup is the ratio of sequential elapsed time to parallel elapsed time. Relative speedup is the ratio of single node parallel time and multiple node parallel time. The absolute speedup indicates the benefits from parallelizing an application. The relative speedup shows the scaling of multithreading and load balancing overheads in a parallel application, when compared to single node parallel execution.

For the Knary and the SPMD applications, sequential version is not available. Therefore all experiments with these two benchmarks will show only the relative speedup. The speedup for all appaications are shown in Figs. 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12.

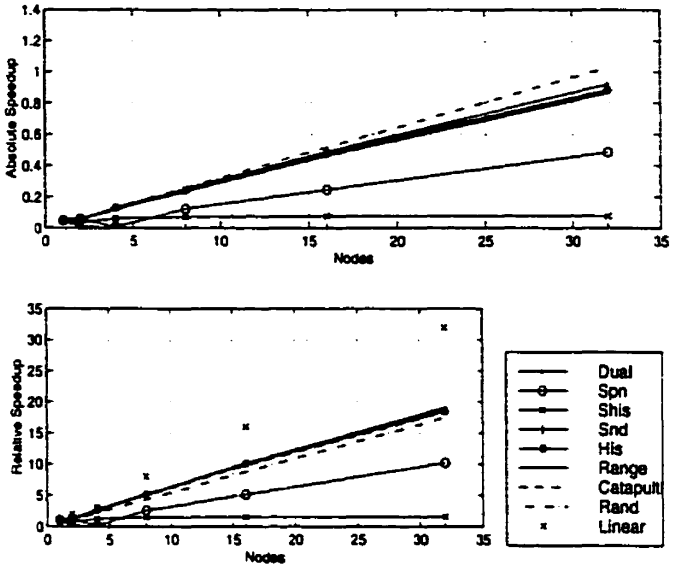


Figure 5.3: Absolute and Relative Speedups for Fibonacci(33)

Figs. 5.13, 5.14 show the scalability of various components that together make up the total elapsed time, such as application thread execution, load balancing overheads, polling overheads, and idle time. Average values for these components on each node are computed, and their scalability compared for the Queens application. The *Rand* balancer scales very well for application thread execution time, and polling overheads. For idle dle time it scales well, but lags behind the *His* balancer.

5.2.3 Parallel Efficiency

- The *Rand* balancer provides relatively good parallel efficiency for both recursive and irregular classes of applications[1]. Furthermore, this efficiency is constant as the number of nodes are varied, suggesting a uniform scaling of parallelism.

  Figs. 5.15, 5.16 show a near constant scaling for the *Rand* baalncer.

[1]Parallel Efficiency is the ratio of absolute speedup and number of nodes in the execution.

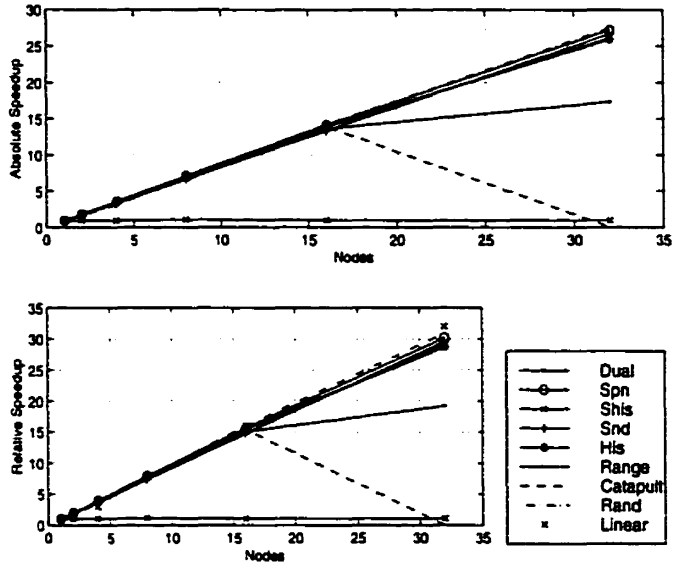


Figure 5.4: Absolute and Relative Speedups for Queens(12)

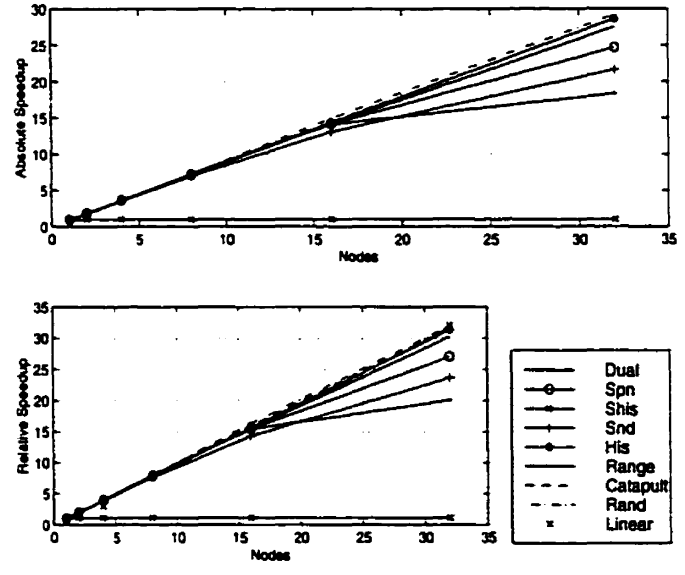


Figure 5.5: Absolute and Relative Speedups for Traveling Salesman Problem(10)

5.2.4 Overheads for Supporting a Multithreaded Environment

The *uni-node support efficiency* or USE factor [84, 156] is the ratio of sequential execution time and the elapsed time for one-node parallel execution. An ideal

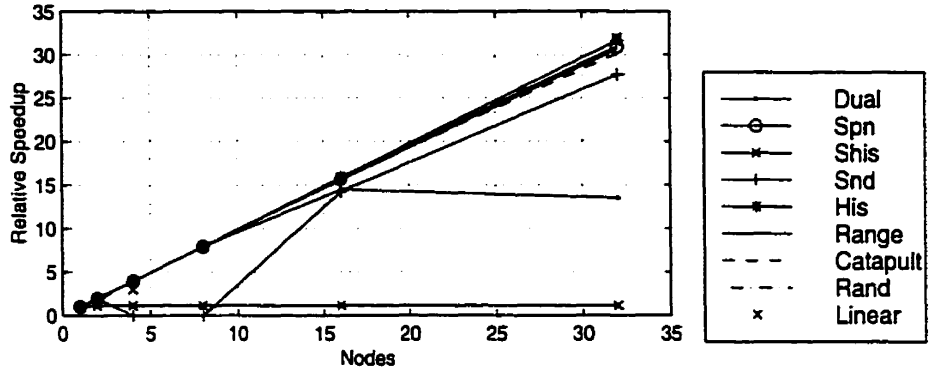


Figure 5.6: Relative Speedup for Knary(7,7,2)

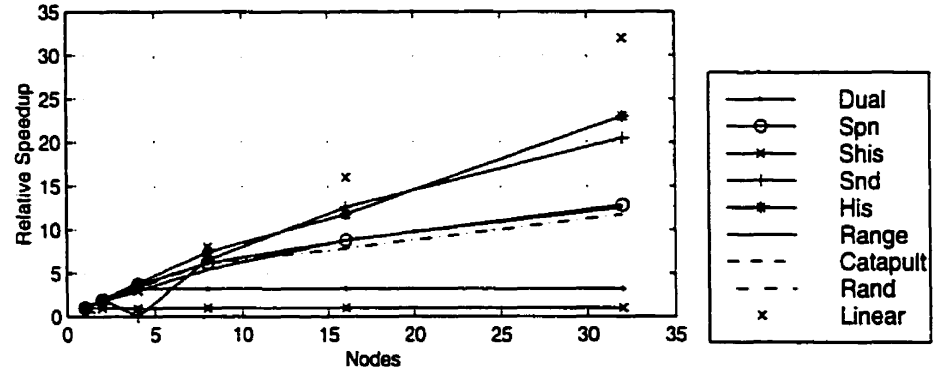


Figure 5.7: Relative Speedup for Knary(2,512,0)

100% use-factor indicates minimum overheads imposed by the multi-threaded environment, and the presence of enough parallelism in the form of threads to hide the latencies of the multi-threaded operations. A unity USE factor also symbolizes good absolute speedup, and indicates the possibility of better and equal load balancing on multiple nodes.

Table 5.6 shows the absolute and relative speedups, and the USE factor for 32 node executions of the complete set of benchmarks. The *Rand* balancer provides better USE factor than the *His* balancer for most of the applications, except for Queens and Tomcatv. In the case of the Queens application, both absolute and relative speedups with the *Rand* balancer are higher than their counterparts for the *His* balancer. This results in better elapsed time for the *Rand* balancer for Queens, but in the case of

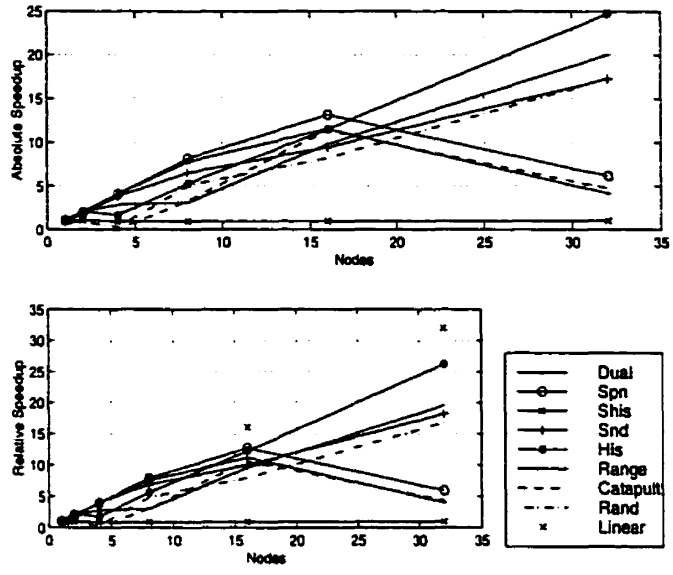


Figure 5.8: Absolute and Relative Speedups for Matrix(1024X1024)

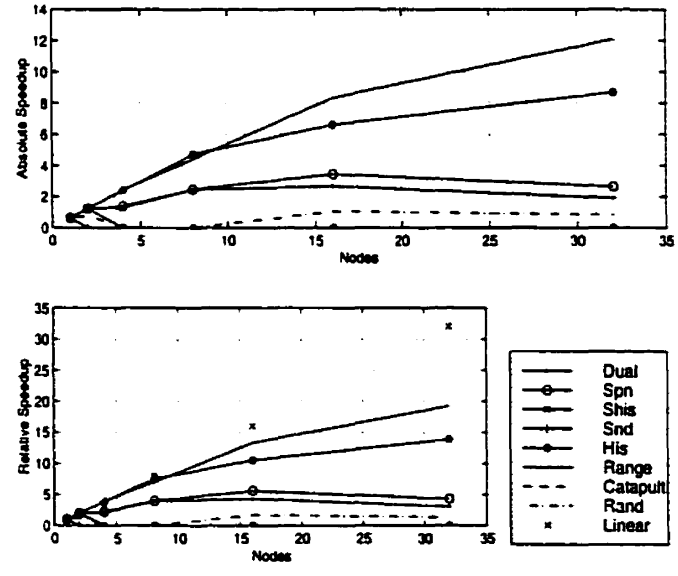


Figure 5.9: Absolute and Relative Speedups for Tomcatv(257)

Tomcatv the performance is much worser than that of the *His* balancer.

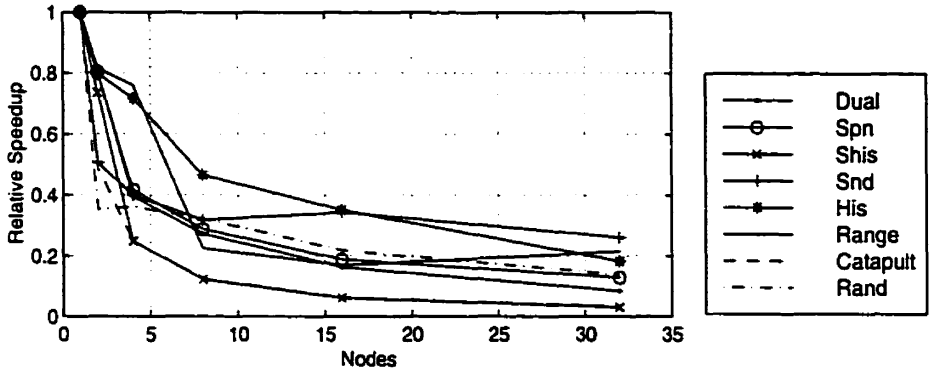


Figure 5.10: Relative Speedup for SPMD(1,1,0)

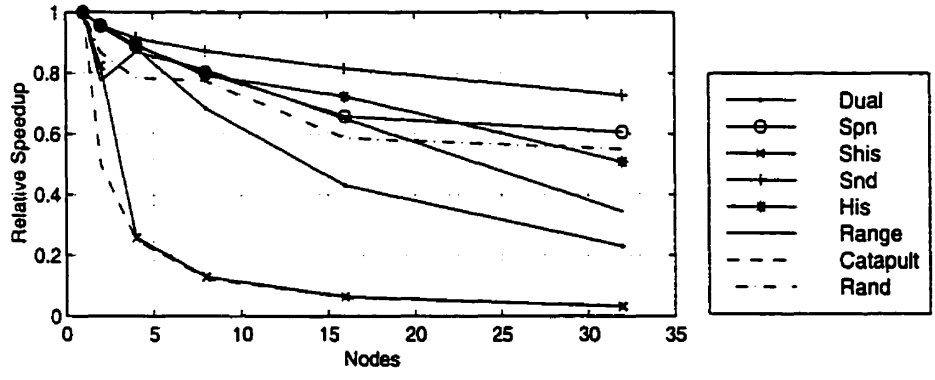


Figure 5.11: Relative Speedup for SPMD(4,4,0)

5.2.5 Distribution of Total Elapsed Time

- The *Rand* algorithm results in nearly equal distribution of both workload and overheads on all the nodes, thereby minimizing system-wide idle time and balancer overheads. One of the reasons for the equal distribution is that, the critical path in the application is executed early in the execution.

  A breakup of the total elapsed time is provided for the Fibonacci, Queens, SPMD, and the Paraffins benchmarks, when they are executed with four different load balancers. Ideally, we would show the bar graph for 32 node executions, but due to space constraints, we are limiting the number of nodes to 8. A description of the profiling strategy is provided in Appendix C.

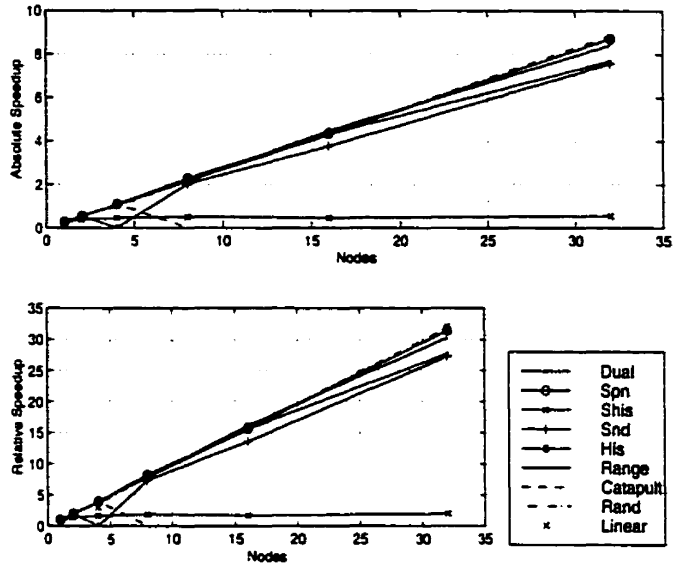


Figure 5.12: Absolute and Relative Speedups for Paraffins(28)

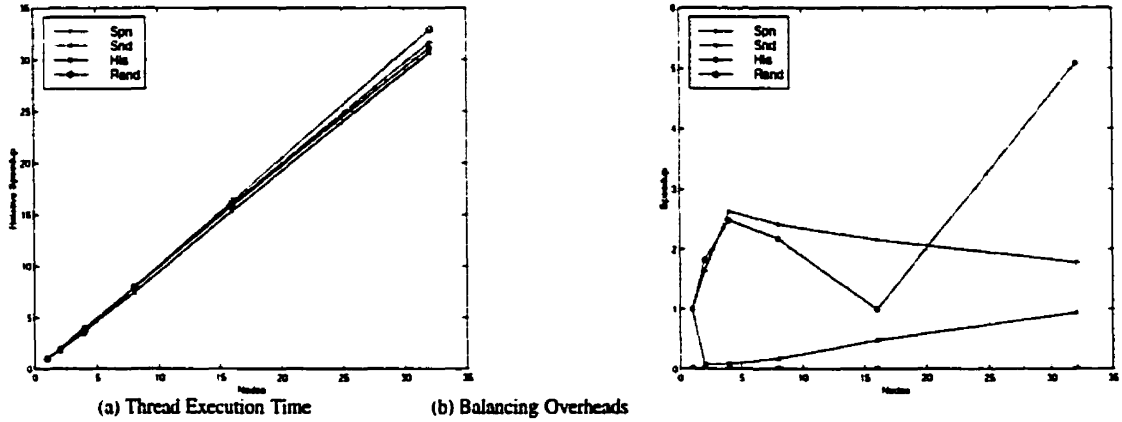
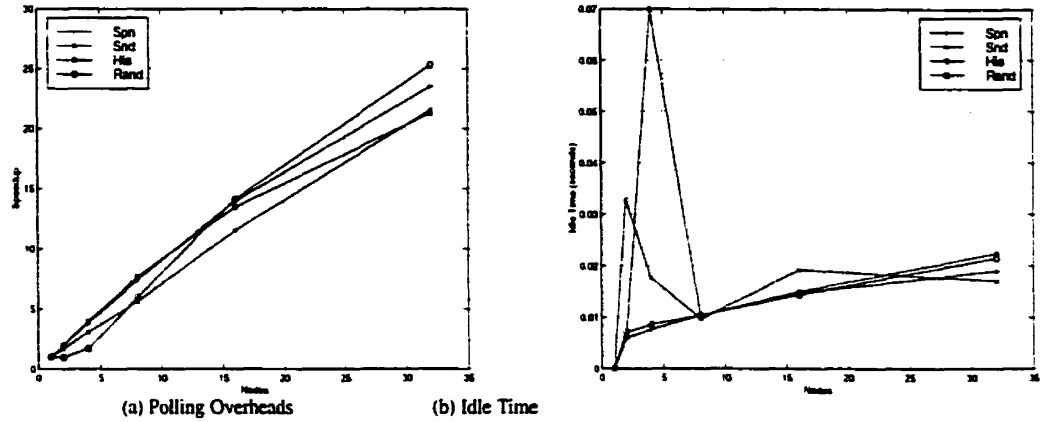


Figure 5.13: Scalability Test for Queens(12)

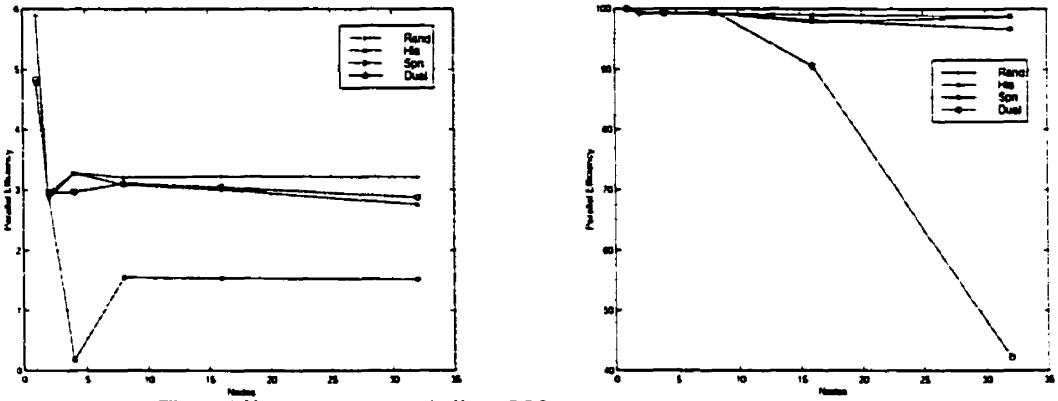
For Fibonacci(33), the high workload in the application causes the *Snd* balancer cause an explosion of load balancing messages resulting in instability in the system. Among the three graphs shown in Figs. 5.17, 5.18, the *Rand* balancer results in most equal distribution of workload and overheads. However for 8 nodes, the *His* balancer provides the best performance.

From Figs. 5.19, 5.20 the results for Queens(12) show the better suitability of the *Spn* balancer for a high workload, fine-grain application. A work-stealing balancer



(a) Polling Overheads (b) Idle Time

Figure 5.14: Scalability Test for Queens(12)



(a) Fibonacci(33) (b) Knary(7,7,2)

Figure 5.15: Parallel Efficiency

involves very less balancer overheads for high workload applications, as the balancer is invoked only when the local node is idle. On the other hand, the *Snd* balancer sends tokens to remote nodes, causing useless load balancing and wasting CPU time on balancer code more often than necessary. The *Rand* balancer performs some token sending, and this causes a slight degradation in performance. In high load situations, the sender component should be restrained to avoid unnecessary load exchanges. While it is important that a balancer distributes workload and overheads as equally as possible on all the nodes in the execution, it is even more important that the balancer should offer scalable performance at higher nodes as well. Results from Fig. 5.4 shows that only the *Rand* balancer is able to provide scalable performance after 8 nodes.

Figs. 5.21, 5.22 show the bar-graph for Knary(7,7,2). The Knary(7,7,2) benchmark is a very high load application, and accordingly all the balancers spend a lot of time

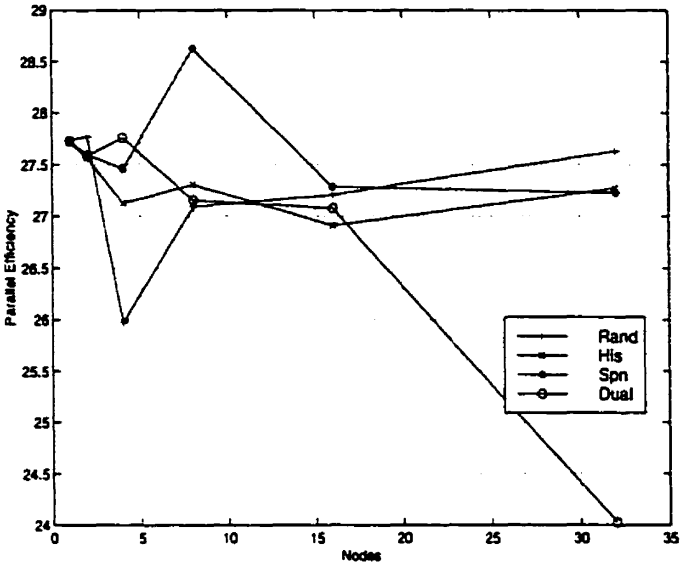
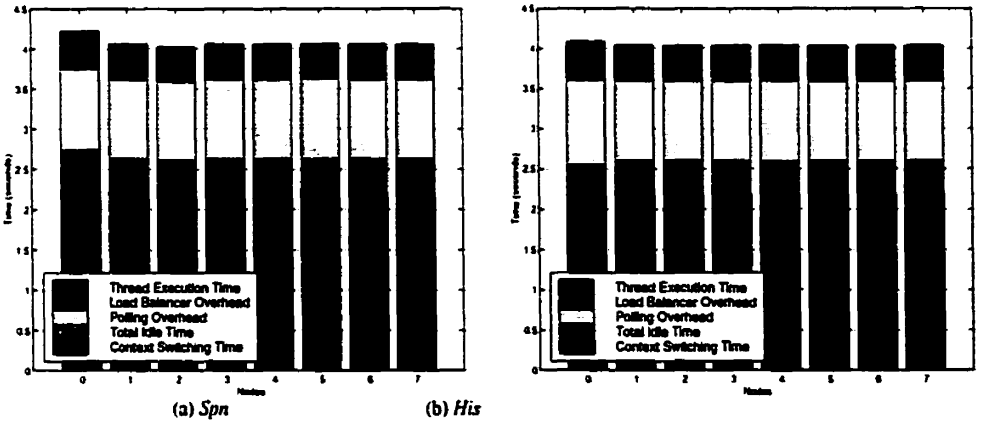


Figure 5.16: Parallel Efficiency for Paraffins(28)



(a) *Spn* (b) *His*

Figure 5.17: A Distribution of Elapsed Time for Fibonacci(33) on 8 nodes

executing application threads, except for the *Snd* balancer which fails due to the high instability in the system.

The SPMD(4,4,0) application is a barrier-synchronized application. All work is created by node 0, and distributed to other nodes in the system in each phase. Therefore it is expected that in Figs. 5.23, 5.24, node 0 spends relatively more time on thread execution. The *Snd* balancer achieves near perfect equal distribution of work and polling overheads. This can be attributed to its fast token distribution

Benchmark	Balancer	Absolute Speedup	Relative Speedup	USE-factor %
Fibonacci(28)	His	0.88	18.49	4.77
	Rand	1.03	17.47	5.89
Queens(12)	His	25.95	28.74	90.29
	Rand	27.47	30.91	88.89
TSP(10)	His	28.66	31.31	91.55
	Rand	29.21	31.89	91.62
Knary(7, 7,2))	His	NA	31.62	NA
	Rand	NA	31.60	NA
Knary(2,512,0)	His	NA	23	NA
	Rand	NA	11.75	NA
Matrix(1024X1024)	His	24.71	26.23	94.18
	Rand	17.29	16.77	103.07
Tomcatv(257)	His	8.69	13.88	62.63
	Rand	0.84	1.37	61.57
SPMD(1,1,0)	His	NA	0.18	NA
	Rand	NA	0.14	NA
SPMD(4,4,0)	His	NA	0.05	NA
	Rand	NA	0.55	NA
Paraffins(28)	His	8.73	31.48	27.72
	Rand	8.84	31.88	27.74

Table 5.6: Absolute and Relative speedups for benchmarks considered in Table 5.1. The USE factor is the *Uni-node Support Efficiency*, and is the ratio of absolute speedup to relative speedup. The numbers above pertain to **32 node** runs.

capability. In fact, node 0 spends less time on thread execution as it spends considerable time disposing extra tokens. The *His* and the *Rand* balancers show expected trends, though the former is able to distribute work from node 0 better. The *Rand* balancer executes more tokens on node 0 as the threshold value is always below the number of tokens in the token queue due to the low load situation, and therefore the sender component is invoked less than the required number of times. The *Spn* balancer performs poorly in this application, as a receiver-initiated balancer is not good for fast token distribution and in low load situations. Further, sending requests with the ring topology in the *Spn* balancer highlights another limiting factor: each subsequent node farther from node 0 gets to execute even lesser tokens than its predecessor.

The irregular nature of the Paraffins application in Figs. 5.25, 5.26 causes wide load

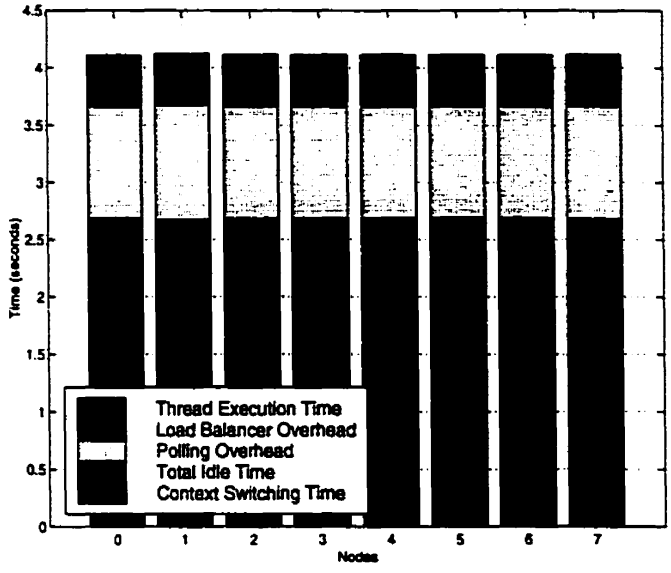
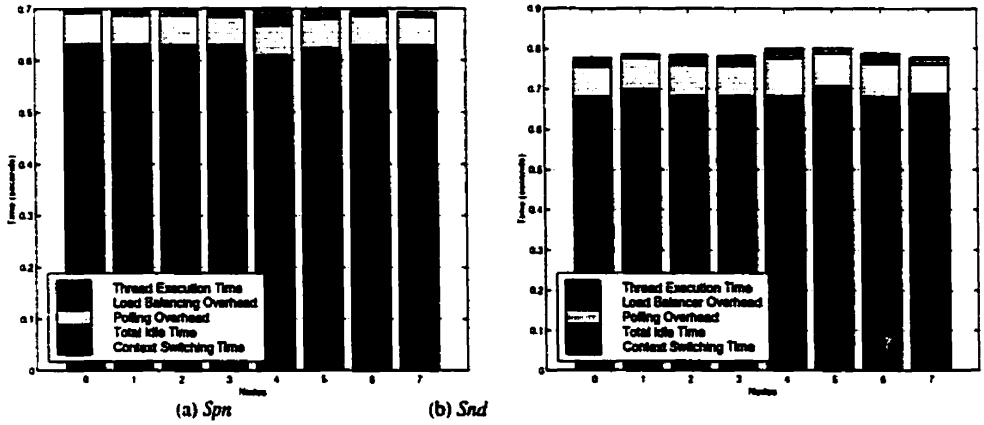


Figure 5.18: A Distribution of Elapsed Time for Fibonacci(33) with Rand Balancer on 8 nodes



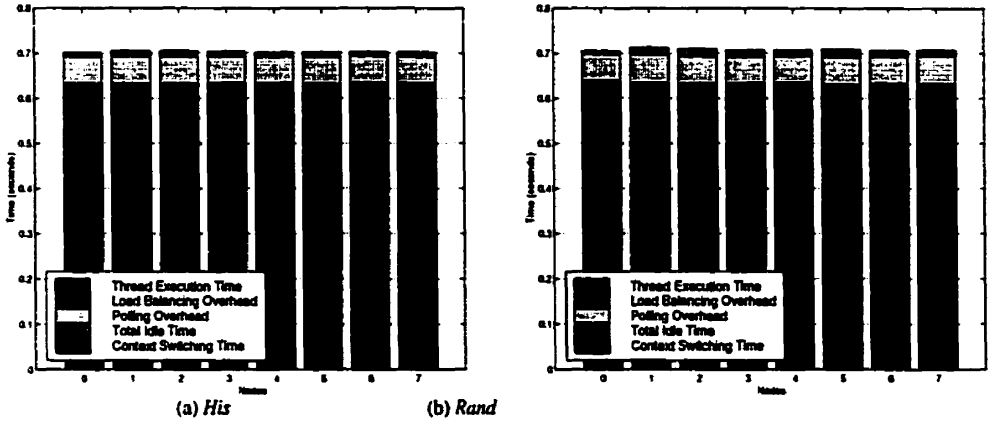
(a) *Spn* (b) *Snd*

Figure 5.19: A Distribution of Elapsed Time for Queens(12) on 8 nodes

imbalances with the *Snd* balancer. The *Rand* balancer provides the best distribution of workload and overheads.

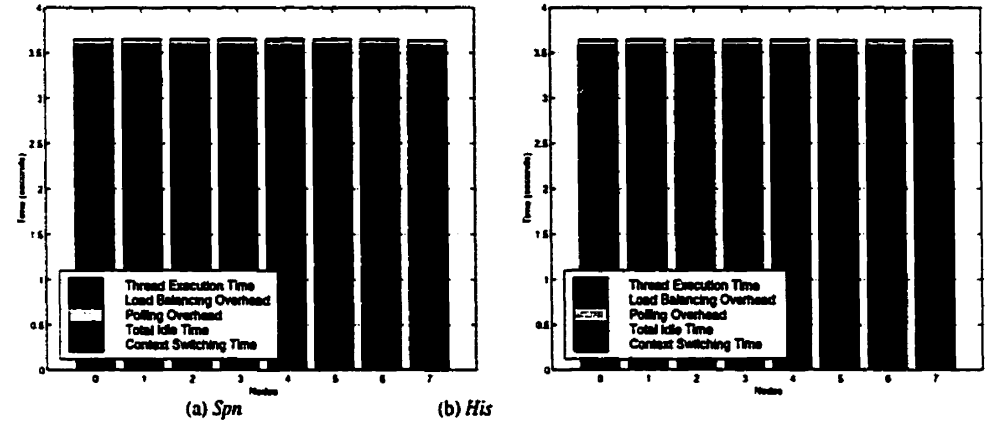
5.2.6 Load State Information and Low Load Applications

- In low load situations, when the amount of parallelism in the application is minimum, using load state information degrades application performance. This is because collecting load state information involves considerable CPU costs, and



(a) *His* (b) *Rand*

Figure 5.20: A Distribution of Elapsed Time for Queens(12) on 8 nodes



(a) *Spn* (b) *His*

Figure 5.21: A Distribution of Elapsed Time for Knary(7,7,2) on 8 nodes

polling overheads. These costs are *avoidable* as the overheads dominate the execution time. They are *unnecessary* because load status of a node changes very rapidly in low load situations, and even the most elaborate information policy cannot guarantee accurate load information. This behavior is demonstrated in Figs. 5.27, 5.28.

5.3 The *Rand* Balancer - A Detailed Study

- The hybrid nature of the *Rand* balancer is the most crucial factor for its good performance. From Tables 5.8, 5.9, it can be seen that for the given set of applications, the order of balancers according to their performance is: *Rand*, *Rand-Hybrid-Noinfo*, *Rand-Rcv-Info*, *Spn*, *His*, *Dual*, where the *Rand* balancer is the best, while the *Dual* balancer yields the poorest performance. This order is determined after ranking

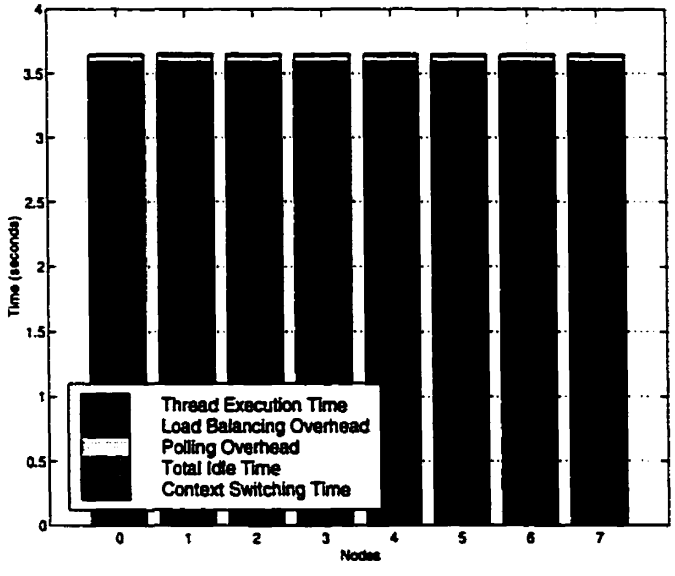
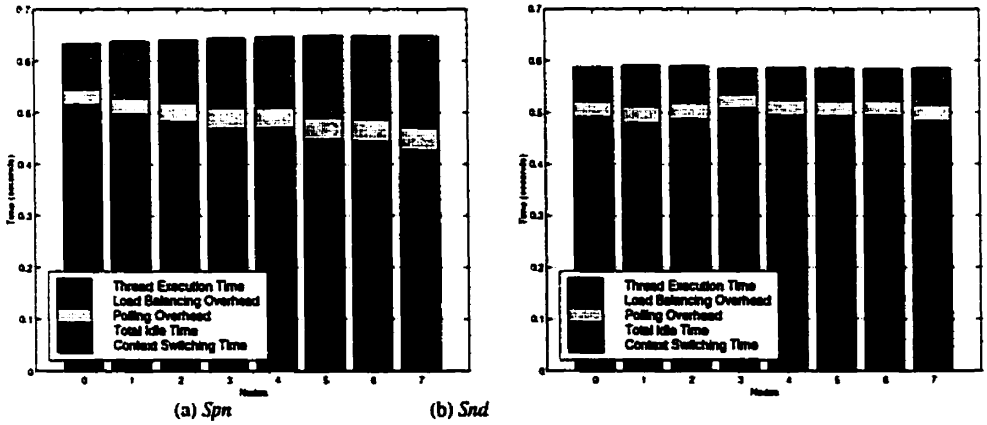


Figure 5.22: A Distribution of Elapsed Time for Knary(7,7,2), Rand Balancer on 8 nodes

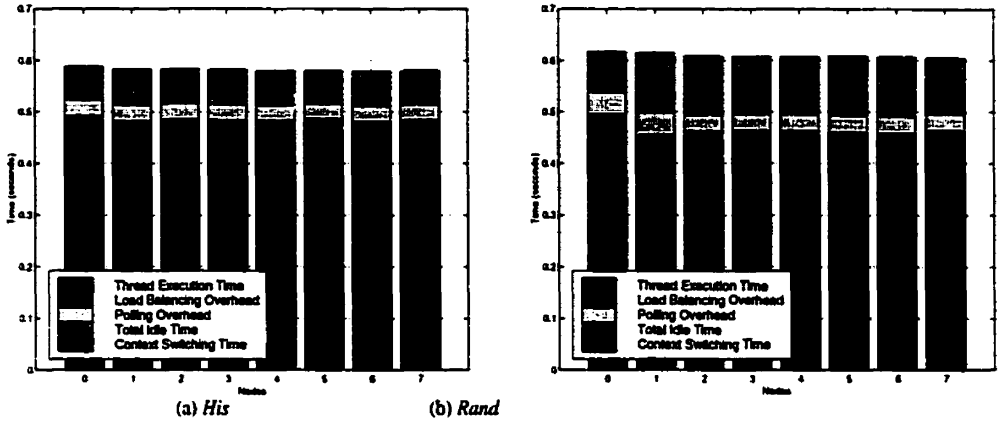


(a) *Spn* (b) *Snd*

Figure 5.23: A Distribution of Elapsed Time for SPMD(4,4,0) on 8 nodes

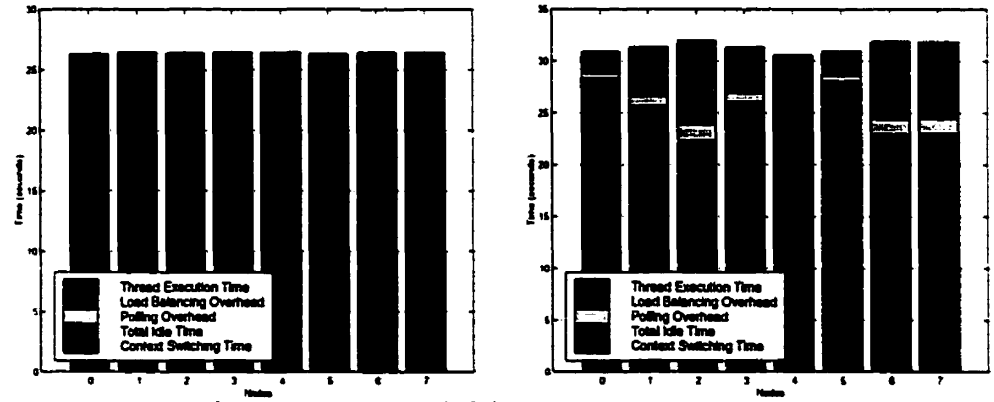
the different balancers for different applications, and then computing the average rank. The *Rand-Hybrid-Noinfo* balancer performs second best after the *Rand* balancer even without the information policy. The *Rand-Rcv-Info* is a receiver-initiated balancer with information policy, but still lags behind the *Rand-Hybrid-Noinfo* balancer in performance.

- Optimum values for threshold to initiate load balancing, and probe limit depend on the number of nodes in the execution.



(a) *His* (b) *Rand*

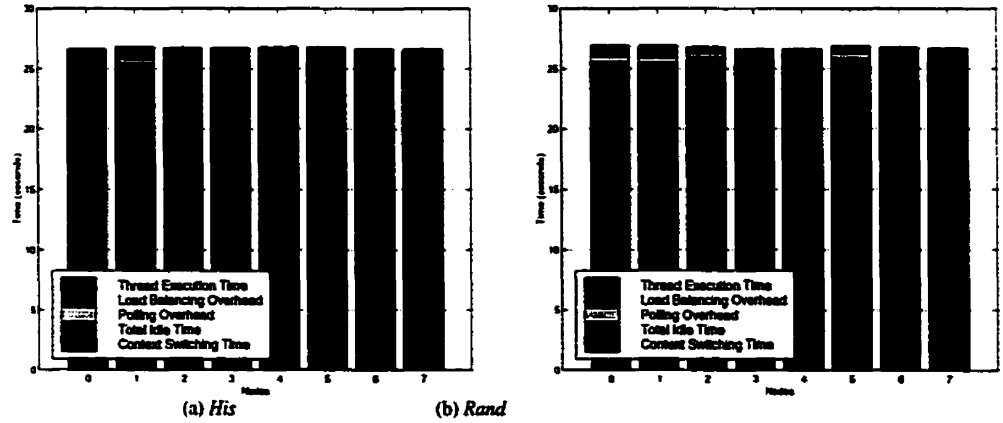
Figure 5.24: A Distribution of Elapsed Time for SPMD(4,4,0) on 8 nodes



(a) *Spn* (b) *Snd*

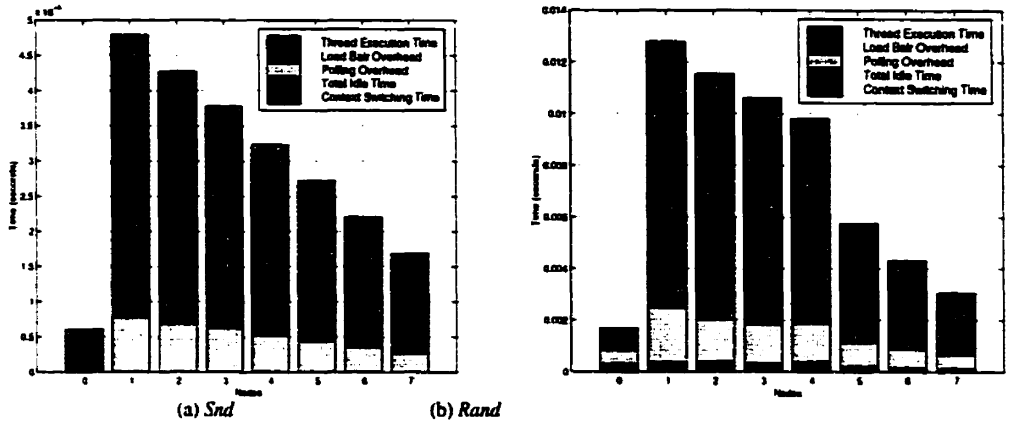
Figure 5.25: A Distribution of Elapsed Time for Paraffins(28) on 8 nodes

Fig. 5.29 shows that the number for load probes (d) derived in section 3.2 reflects the system configuration well. The number of load probes when computed from the



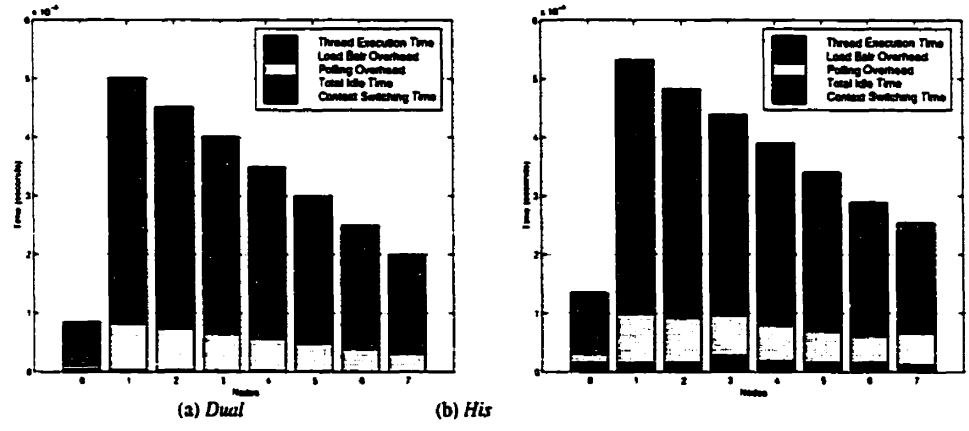
(a) *His* (b) *Rand*

Figure 5.26: A Distribution of Elapsed Time for Paraffins(28) on 8 nodes



(a) *Snd* (b) *Rand*

Figure 5.27: Distribution of Elapsed Time for Fibonacci(6)



(a) *Dual* (b) *His*

Figure 5.28: Distribution of Elapsed Time for Fibonacci(6)

number of nodes in the execution, balances the trade-off between overheads from load probes, and better accuracy of the execution. Queens(12) is a very fine-grain application with high workload. The benefits of computing the number of load probes in this manner will be more apparent for other applications.

Fig. 5.30 shows the impact of varying the upper threshold, i.e. the threshold on crossing which the sender component in the balancer is initiated.

- Sending load probes to collect global load information is beneficial in the receiver mode, but not in the sender mode. Therefore, it is preferable to initiate load probes in the receiver mode, and use this information in the sender mode. From Tables 5.8, 5.9, the performance of the *Rand-Snd-Info* balancer compares very poorly against that of the *Rand* and the *Rand-Rcv-Info* balancers.

Benchmark	Rand-Hybrid-Info	Rand-Hybrid-NoInfo	Rand-Snd-Info	Rand-Rcv-Info
Fibonacci(33)	1.02	1.15	OF	1.22
Queens(12)	0.165	0.168	0.19	0.167
TSP(10)	0.27	0.29	0.31	0.28
Knary(7,7,2)	0.906	0.913	OF	0.907
SPMD(4,4,0)	0.79	0.77	1.74	0.82
Paraffins(28)	6.46	6.51	8.33	6.52

Table 5.7: Performance of the Rand balancer in different modes. The Rand-Hybrid-Info balancer is a hybrid rand balancer that used load state information. The other modes are self explanatory. All measurements are in **seconds**, and correspond to total elapsed times on **32 nodes**. These numbers include the time spent on profiling code.

Benchmark	Dual	Spn	Rand-Rcv-Info	Snd	Rand-Snd-Info	His
Fibonacci(33)	1.14	1.14	1.22	OF	OF	1.19
Queens(12)	0.24	0.167	0.167	0.171	0.19	0.176
TSP(10)	0.43	0.32	0.28	0.36	0.31	0.28
Knary(7,7,2)	2.13	0.93	0.907	1.037	OF	0.908
SPMD(4,4,0)	1.9	0.72	0.82	0.63	1.74	0.86
Paraffins(28)	7.43	6.55	6.52	7.54	8.33	6.54

Table 5.8: Performance comparison between the receiver-initiated, sender-initiated and hybrid balancers and their counterparts using the randomizing algorithm. All measurements are in **seconds** and represent **32 node** runs.

Benchmark	Rand	Rand-Hybrid-Noinfo	Rand-Rcv	Rand-Hybrid	Rand-Hybrid-Piggyback
Fibonacci(33)	1.02	1.15	25.34	1.18	36.05
Queens(12)	0.165	0.168	12.42	4.74	5.05
TSP(10)	0.27	0.28	0.78	0.29	1.82
Knary(7, 7,2)	0.906	0.913	25.22	0.965	28.95
SPMD(4,4,0)	0.79	0.775	0.83	0.70	14.18
Paraffins(28)	6.46	6.51	43.22	7.08	90.65

Table 5.9: Performance comparison between different versions of the Rand balancer. This table shows the relevance of the randomization policy, information policy, and receiver/sender/hybrid policy, which together make up the Rand balancer. All measurements above are in **seconds**, and are based on **32 node** runs.

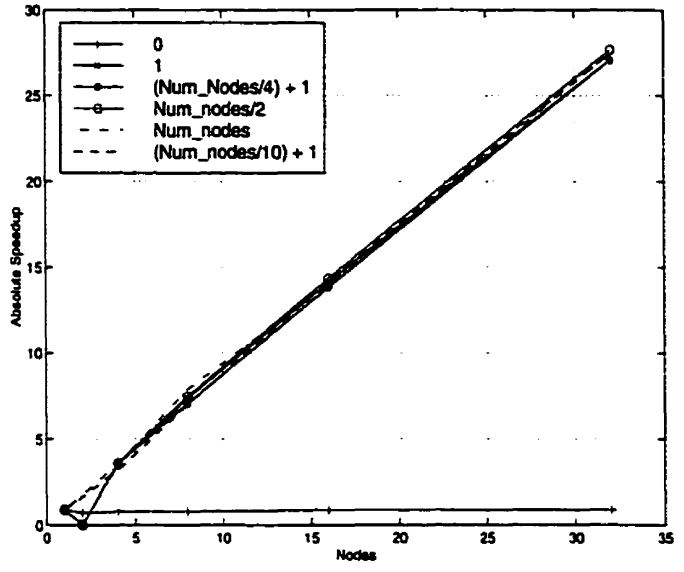
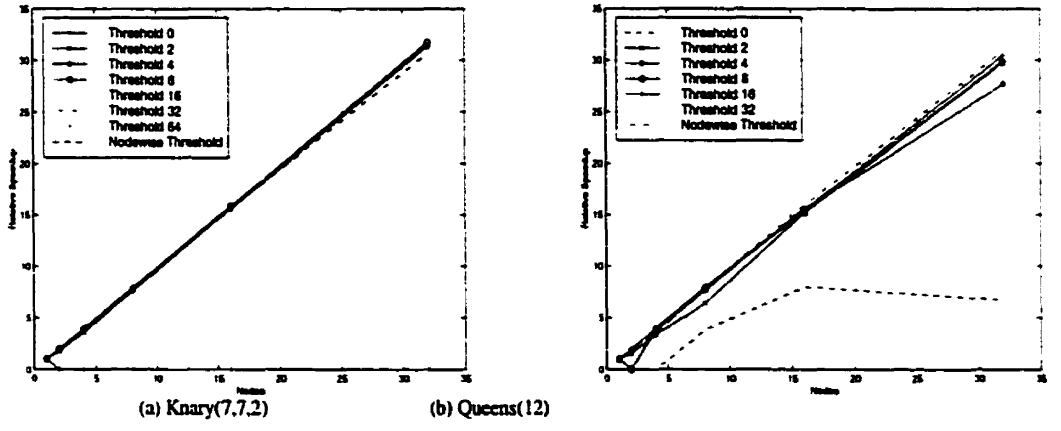


Figure 5.29: Performance of Queens(12) while varying the number of random probe destinations



(a) Knary(7,7,2) (b) Queens(12)

Figure 5.30: Effect of Load balancing with Rand Balancer. Load balancing threshold is varied in the balancer.

- The sender mode in the hybrid balancer reacts very adversely with inaccurate global load state information, often causing instability. On the other hand, accurate load information improves performance and robustness significantly.

- A work-stealing randomizing balancer using global load information outperforms a hybrid balancer using history information. From Table 5.8, the *Rand-Rcv-Info* balancer outperforms the *His* balancer by a convincing margin.

- A hybrid, randomizing balancer - *Rand-Hybrid*, that does not use load state information at all (no load probes) provides the second-best performance for barrier-synchronized applications. This relatively better performance when compared to the *Rand* balancer can be attributed to the avoidance of polling overheads in collecting load state information.

- A simple, randomizing, work-stealing balancer with no access to global load information results in unacceptable performance for Threaded-C applications, as shown in Table 5.9. The *Rand-Rcv* balancer is based on a simple work-stealing algorithm, and chooses load destinations by executing a randomizing function. This policy does very poorly for Threaded-C applications.

The scalability of different versions of the *Rand* balancer are compared in Figs. 5.31, 5.32, 5.33. The *Rand-hybrid-Info* balancer in these figures refers to the *Rand* balancer. The *Rand* balancer leads with better performance for all recursive and irregular applications, confirming the utility of all features that combinedly form the *Rand* balancer. For the irregular application Paraffins, the *Rand* balancer initially lags behind the *Rand-Hybrid-Noinfo* and *Rand-Rcv-info* balancers. this is because the sender component of the *Rand* balancer chooses destinatoins based on load information collected previously by the receiver component. Due to the irregular nature of the Paraffins application, this load information is not valid anymore, and as a result useless load balancing occurs. However, as the number of nodes are increased, the *Rand* balancer starts dominating the other balancers.

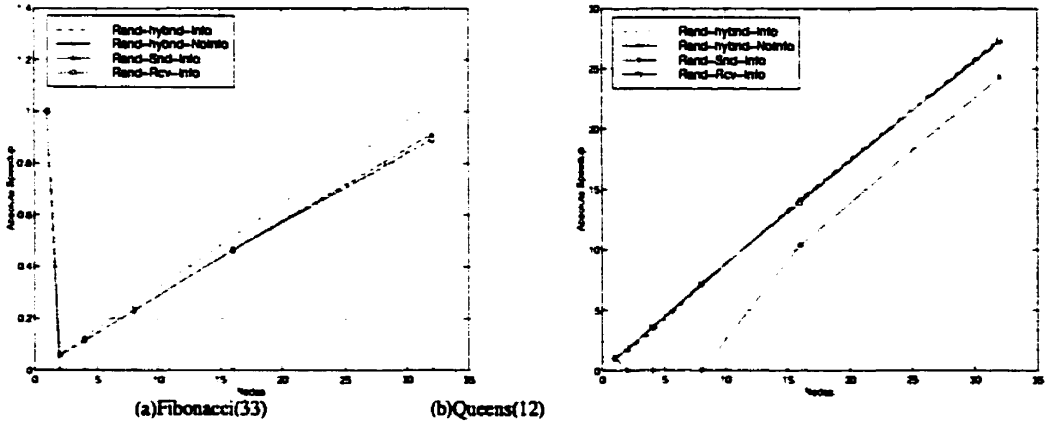
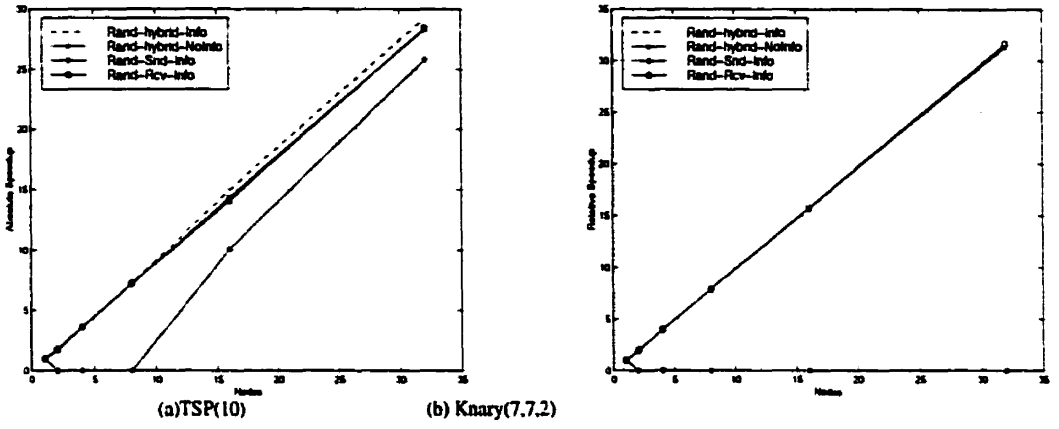


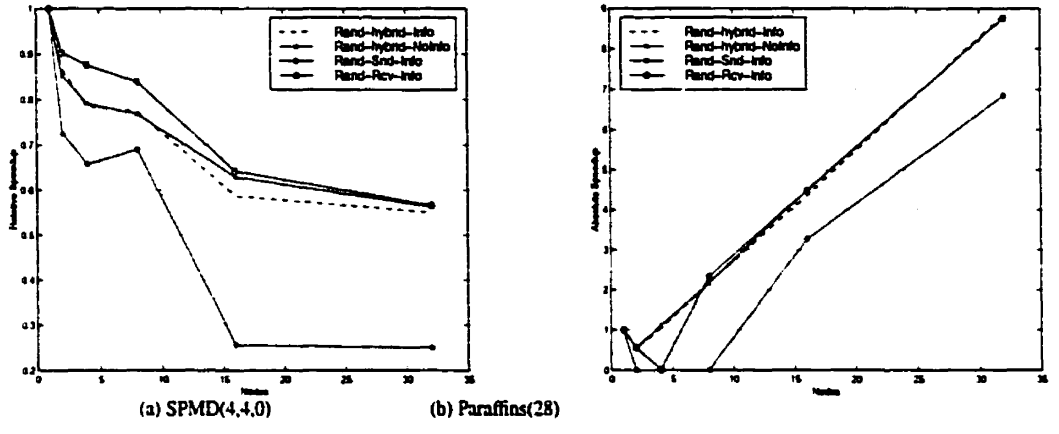
Figure 5.31: Performance of different randomizing policies

The effect of information policy in the *Rand* balancer on application performance is shown in Fig. 5.34. Different levels of information availability are considered.



(a)TSP(10) (b) Knary(7,7,2)

Figure 5.32: Performance of different randomizing policies

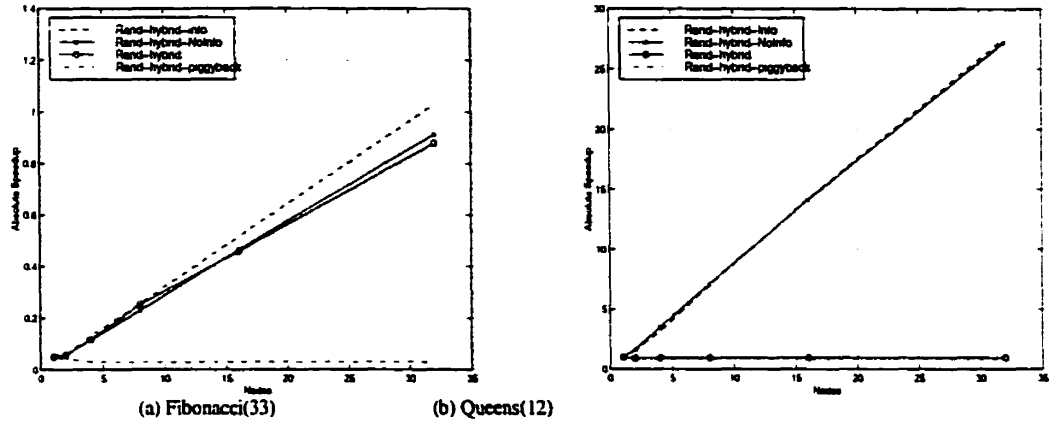


(a) SPMD(4,4,0) (b) Paraffins(28)

Figure 5.33: Performance of different randomizing policies

They are: no load information and no load probes (*Rand-Hybrid*); limited load information available only by piggybacking on regular load balancing messages (*Rand-hybrid-Piggyback*); no load information in the sender component, though load probes are used in receiver component (*Rand-Hybrid-NoInfo*),and finally the regular *Rand* balancer (*Rand-hybrid-info*). The *Rand* balancer outperforms other versions of itself.

Table 5.10 shows the profile data for an 8-node execution for Fibonacci(33). The results shown here help us better understand the utility of the load balancer in impoving the performance for a particular application. Locality is the ratio of tokens that are generated locally and locally consumed over all the tokens consumed locally. Migration is the ratio of tokens that are migrated to remote nodes over all the tokens generated and received on a node.

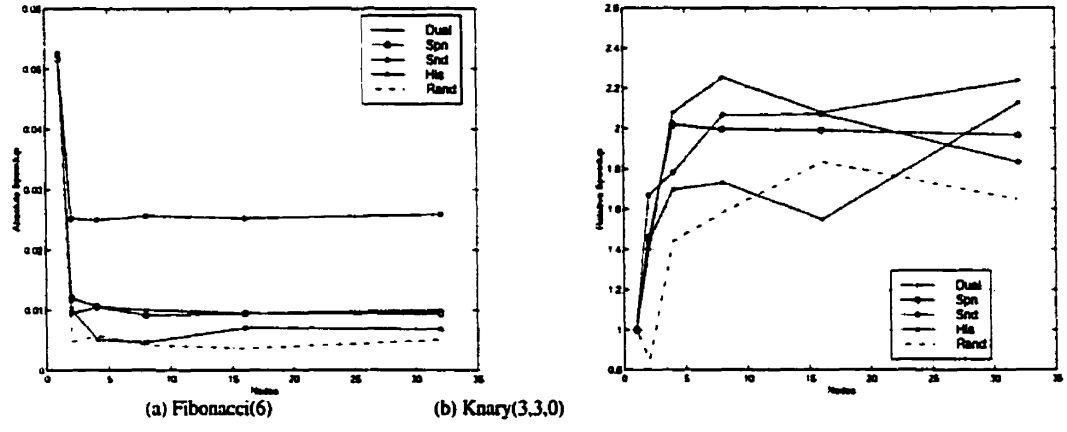


(a) Fibonacci(33) (b) Queens(12)

Figure 5.34: Effect of Information policy in the Rand balancer

5.4 Other Balancers

When the *Rand* balancer does not perform well, an in-depth study is performed on alternative balancers.



(a) Fibonacci(6) (b) Knary(3,3,0)

Figure 5.35: Relative performance of balancers at low workloads (a)Low loads, very fine grain threads (b) Low loads, grain size 100μs, polling interval 50μs

- A sender-initiated balancer (*Snd*) is the best choice in two scenarios: barrier-synchronized applications, and very fine-grain applications at low input workloads. This is shown in part(a) of Fig. 5.35, and Figs. 5.10, 5.11.

- A simple receiver-initiated balancer (*Dual*) is preferable for fine-grain applications with modest thread granularities and at very low input workloads, as shown in part(b) of Fig. 5.35. This policy uses a ring as logical topology to send requests, and to receive workloads.

Balancer	Attribute	Nodes							
		0	1	2	3	4	5	6	7
Rand-Hybrid-Info 4.1s	Requests	155	237	200	188	215	205	211	132
	Idle Periods	27	23	13	21	14	23	24	13
	Extra Tokens	–	–	–	–	–	–	–	–
	Locality %	99	99	99	99	99	99	99	99
	Migration %	0.04	0.03	0.01	0.02	0.02	0.02	0.02	0.02
Rand-Hybrid-NoInfo 4.48s	Requests	166	262	237	214	206	187	182	142
	Idle Periods	40	27	24	14	21	16	21	14
	Extra Tokens	887	872	443	444	434	497	490	453
	Locality %	99	99	99	99	99	99	99	99
	Migration %	0.07	0.06	0.03	0.03	0.03	0.04	0.03	0.03
Rand-Hybrid 4.12s	Requests	266	253	154	190	173	198	173	202
	Idle Periods	16	23	18	14	29	24	18	21
	Extra Tokens	5201	5351	2640	2635	2616	2869	2732	2630
	Locality %	99	99	99	99	99	99	99	99
	Migration %	0.40	0.41	0.18	0.18	0.18	0.20	0.19	0.18
Rand-Rcv-Info 4.49s	Requests	158	280	221	195	207	186	161	136
	Idle Periods	32	32	20	14	24	21	27	21
	Extra Tokens	116	118	54	59	80	81	74	62
	Locality %	99	99	99	99	99	99	99	99
	Migration %	0.01	0.01	0.00	0.00	0.01	0.01	0.01	0
Dual 4.05s	Requests	67	66	68	77	80	77	77	73
	Idle Periods	20	18	23	16	15	17	21	17
	Extra Tokens	59	59	58	70	74	70	67	63
	Locality %	99	99	99	99	99	99	99	99
	Migration %	0.01	0.00	0.01	0.01	0.01	0.01	0.01	0
Spn 4.11s	Requests	136	146	145	153	144	139	140	140
	Idle Periods	21	24	25	25	23	26	34	22
	Extra Tokens	75	76	72	78	74	70	66	77
	Locality %	99	99	99	99	99	99	99	99
	Migration %	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
His 4.24s	Requests	399	181	329	357	371	391	446	371
	Idle Periods	49	37	56	56	64	47	57	45
	Extra Tokens	–	–	–	–	–	–	–	–
	Locality %	99	99	99	99	99	99	99	99
	Migration %	0.03	0.01	0.02	0.02	0.02	0.02	0.03	0.02

Table 5.10: A study of load balancer behavior for Fibonacci(33)

- History information (*His*) performs better than global load information (*Rand*) for regular applications. This is shown in Fig. 5.8, 5.9.

5.5 Program Behavior

We designed a spectrum of experiments to understand application behavior with different load balancers. Wherever speedup curves are mentioned, we mean absoulte speedup, with the number of processors along the x-axis, and the absolute speedup along the y-axis.

- The application and runtime parameters that determine performance are: programming model, parallelism grain size, input workload, polling interval, number of nodes, load balancer strategy. Load balancer related issues like balancer policies, logical topology, quality of load state information, message complexity, CPU time spent on executing the balancer code play a significant role. Other system related factors are: ratio of CPU speed to network speed, network bandwidth, network topology, network interface in the runtime system.

- For irregular and recursive applications, load state information outperforms history information. However for regular applications, load state information actually degrades performance.

5.5.1 Transition Point and Peak Point

- A *transition point* is the absolute speedup for a 2-node execution. It represents the transition from one-node execution to parallel execution. The performance at this point is indicative of the amount of parallelism in the application, and how well the system is able to exploit the available parallelism. The higher the absolute speedup at this point, the better is the suitability of the balancer for this application. This also provides an early trend to the relative performance of different load balancers for this application, for any workload and for a reasonable number of nodes. An upward or downward slope of the curve at this point reflects on the balance between load balancer overheads and benefits from parallel execution.

  Fig. 5.36, part(a) shows the transition and peak points for Fibonacci(12). At 2 nodes, the *Dual*, *His*, *Rand*, *Snd* balancers are in decreasing order of performance.

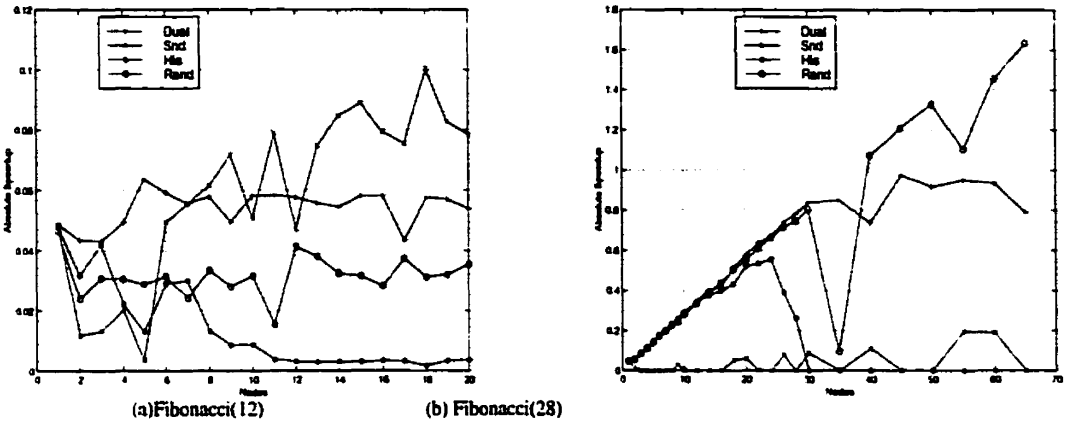


Figure 5.36: Comparision of speedup against overheads with the increase in the number of Nodes for Fibonacci

Part(b) shows the absolute speedups for Fibonacci(28). Until 32 nodes, the *Dual* balancer leads the rest of the balancers. The *Dual* is followed by *His* (until 14 nodes), *Rand*, and *Snd*. This is according to the predictions made from the transition point from part(a). As explained before, the transition point does not cover scalability, and is valid for only a modest number of nodes. The *Rand* balancer is seen here dominating from 40 nodes onwards.

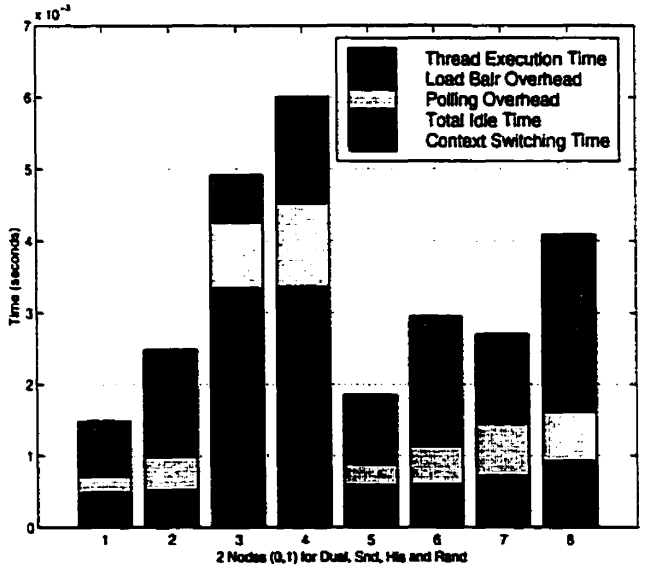
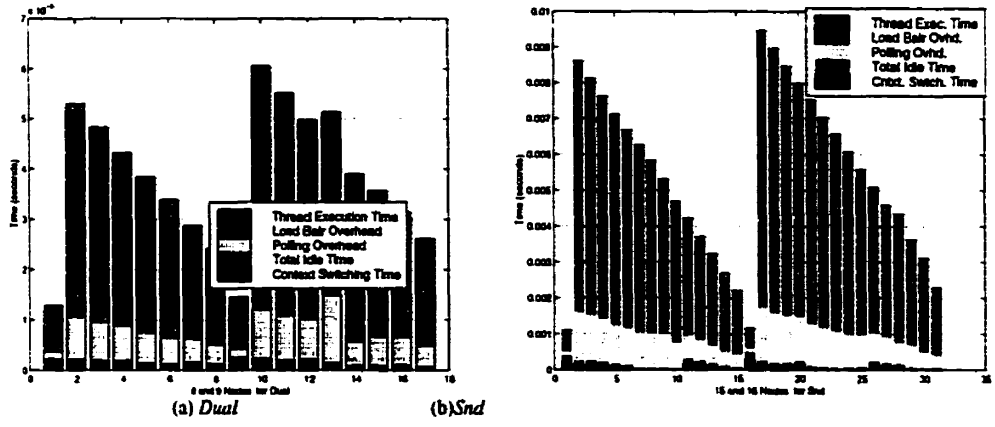
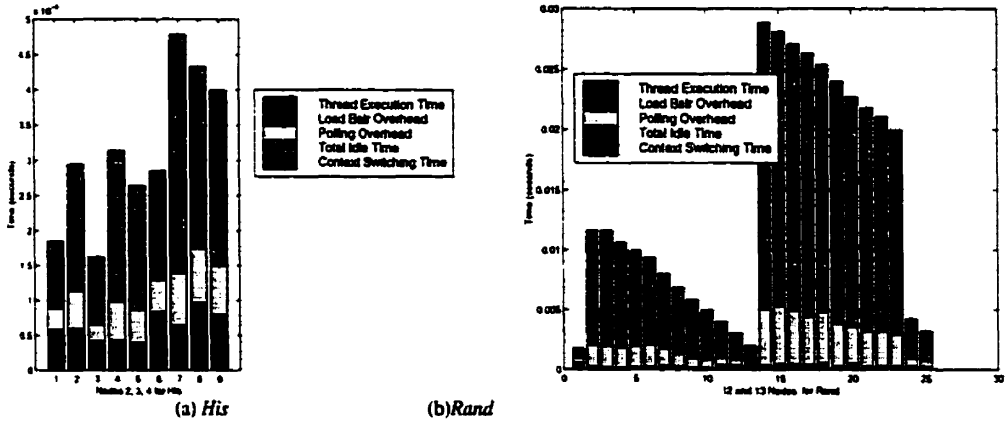


Figure 5.37: A comparision of Transition points for different balancers for Fibonacci(12).



(a) *Dual* (b)*Snd*

Figure 5.38: Peak performance points for Fibonacci(12)



(a) *His* (b)*Rand*

Figure 5.39: Peak performance points for Fibonacci(12)

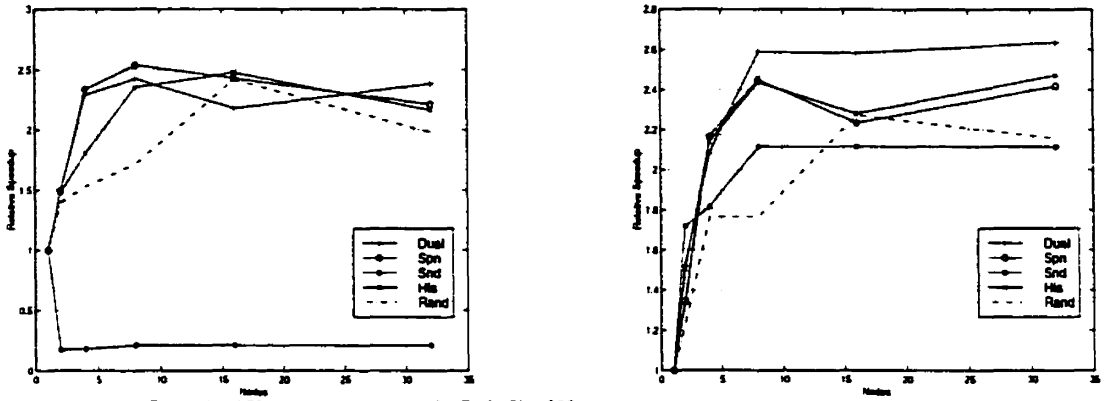
- A *peak point* is the highest point on the speedup curve. After this point, the speedup curve usually becomes constant or dips downward. The higher the number of processors at this point, the better the scalability of the balancer. This point predicts the same scalability trend of the balancer for any workload.

  From Fig. 5.36, part(a), the peak points according to decreasing order belong to *Snd*, *Rand*, *Dual*, and *His* balancers. This pattern can be observed in part(b) of this figure. Th *Rand* balancer is not only scalable, but also stable, unlike the *Snd* balancer which fails after creating instability in the system due to load thrashing.

5.5.2 Effect of Grain Size and Polling Interval

- Increase in grain size improves application performance as long as there is enough parallelism to be exploited. Similarly increase in workload represents high amount

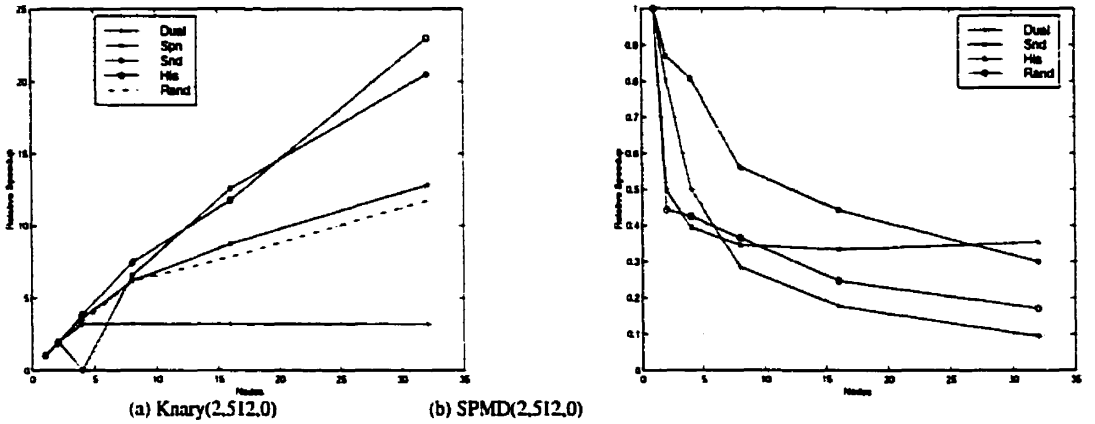
of parallelism in the application. This provides a better amortization of load balancer overheads and results in better load balancer performance. At low workloads, a balancer with least overheads performs best.



(a) Grain Size 150 μs (b) Grain Size 200 μs

Figure 5.40: Performance comparision at low loads at polling interval of 50μs for Knary(3,3,0)

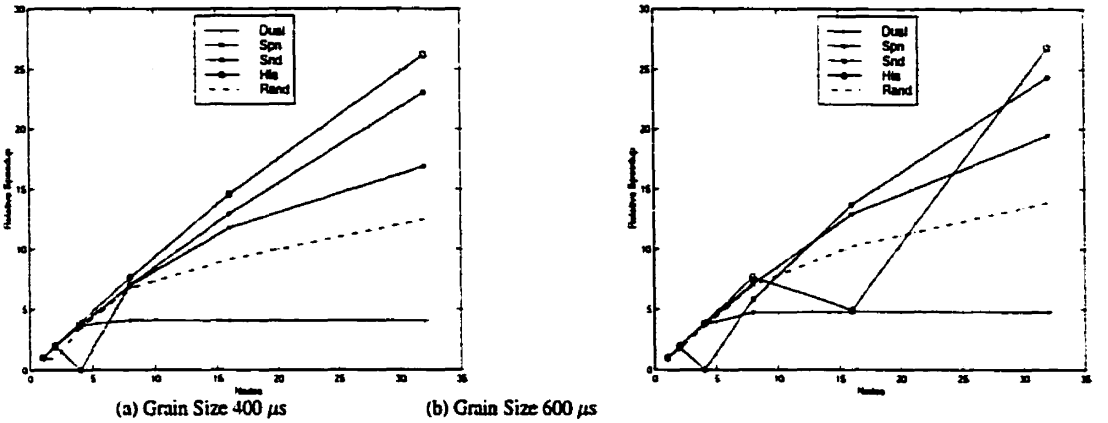
Fig. 5.40 shows that with increase in grain size, the *His* balancer outperforms the *Spn* balancer. This is due to the fact that the increased grain size dominates the balancer overheads of the *His* balancer.



(a) Knary(2,512,0) (b) SPMD(2,512,0)

Figure 5.41: Relative Speedup with grain size 200μs

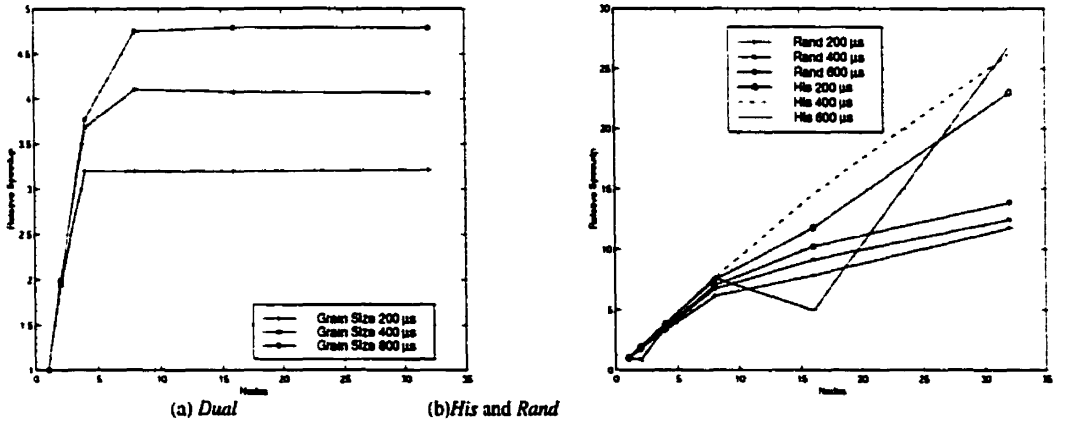
In Fig. 5.41, the *His* balancer proves to be more scalable than the *Snd* balancer. This result also highlights the ability of the *His* balancer to distribute tokens very fast. However, part(b) of Fig. 5.41 shows that for the same workload, the low load situation for each phase in a barrier-synchronized application suits a pure sender-initiated balancer better.



(a) Grain Size 400 μs (b) Grain Size 600 μs

Figure 5.42: Relative Speedup for Knary(2,512,0)

In Fig. 5.42, doubling the token grain size slightly improves corresponding numbers for speedup achieved. It should be noted that with increase in grain size, and no corresponding increase in workload, the amount of exploitable parallelism in the application dwindles.

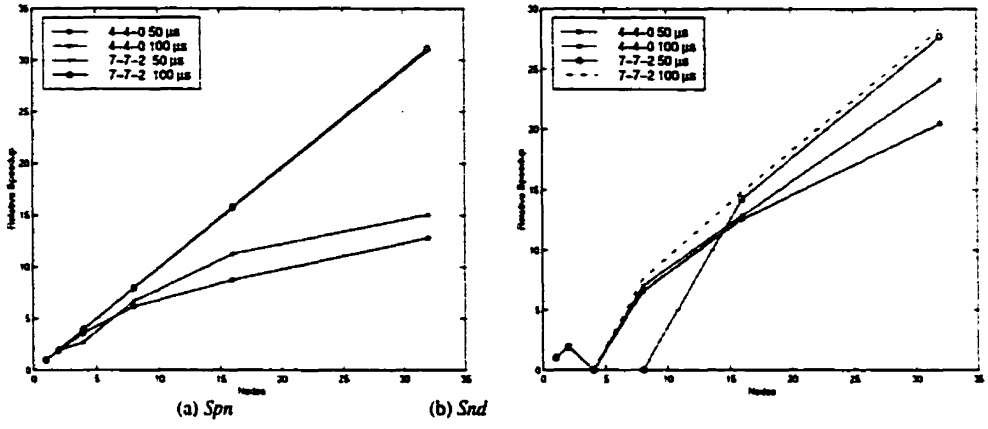


(a) *Dual* (b)*His* and *Rand*

Figure 5.43: Performance of Knary(2,512,0) for different grain sizes

Fig. 5.43 shows that increase in grain size improves performance. However, this improvement is not much significant for the *Dual* balancer as it is for the other balancers partly due to the fact that ring topology is not equipped for fast token distribution, and secondly becasue of the receiver-initiated policy of the *Dual* balancer finds it difficult to locate a rich node in a low load situation. Only a few nodes close to the rich nodes end up consuming all the tokens. Therefore, after achieving certain initial speedup, the curve flattens out.

- At lower workloads, polling interval has a significant impact on application performance. However at higher workloads, the effect of polling interval is not as relevant as at low workloads.

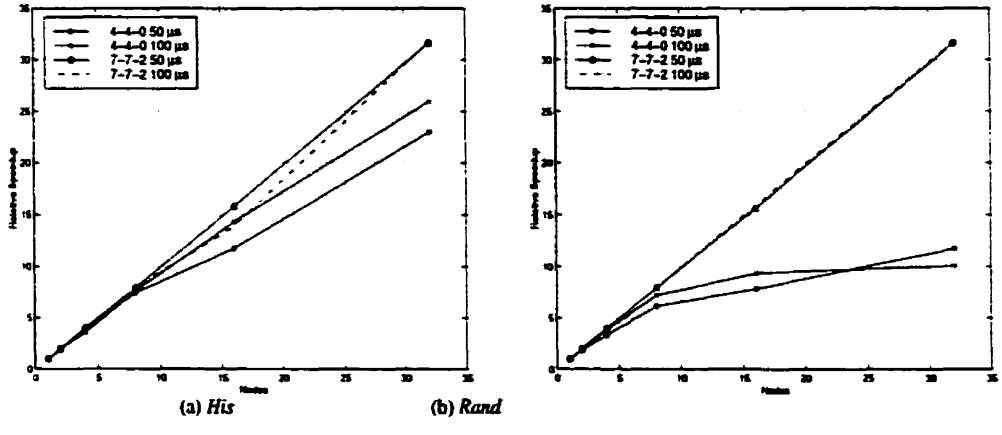


(a) *Spn* (b) *Snd*

Figure 5.44: Performance of Knary(4,4,0) and Knary(7,7,2)

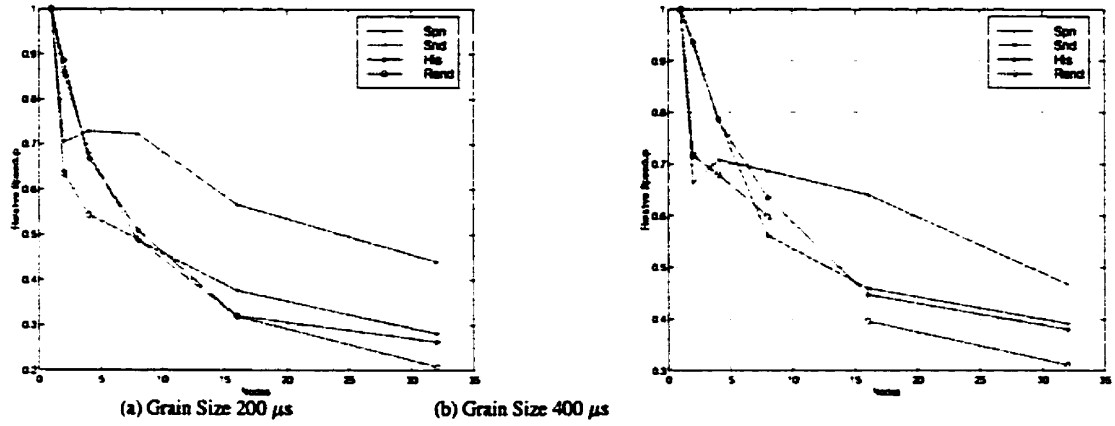
Fig. 5.44 shows the equation between workload and polling interval, for the *Spn* and *Snd* balancers. At small workloads, increase in polling interval improves performance due to the minimization of network access overheads. However, this does not make much difference at high workloads. There is a slight difference for high workloads with increase in polling interval for the *Snd* balancer, because of the dependence of the *Snd* balancer on token disposal across the network.

- Higher polling interval improves performance due to low polling overheads, unless the balancer depends significantly on remote communications (for instance the *Rand* balancer).



(a) *His* (b) *Rand*

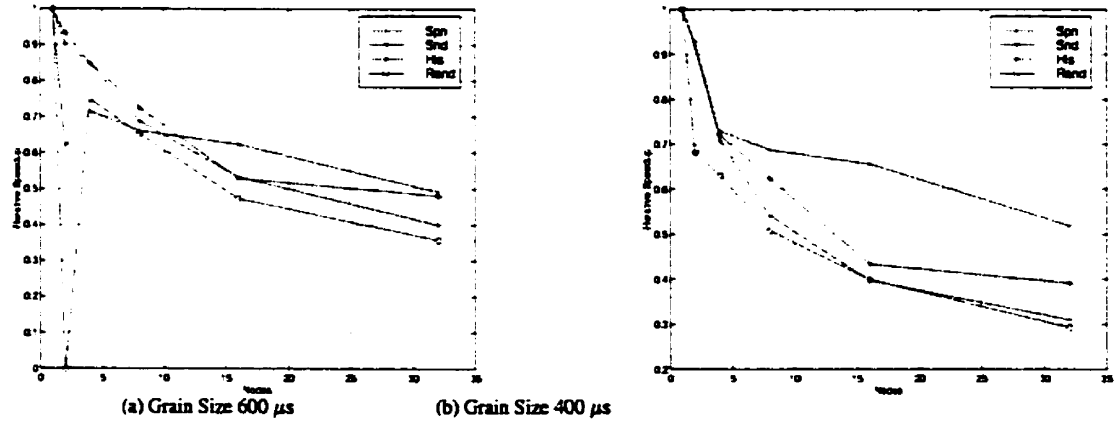
Figure 5.45: Performance of Knary(4,4,0) and Knary(7,7,2)



(a) Grain Size 200 μs (b) Grain Size 400 μs

Figure 5.46: Performance of SPMD(3,3,0) at polling interval of 50 μs

Fig. 5.46 demonstrates similar behavior for barrier-synchronized applications with increase in grain size. Corresponding numbers for speedup have increased with increase in grain size.



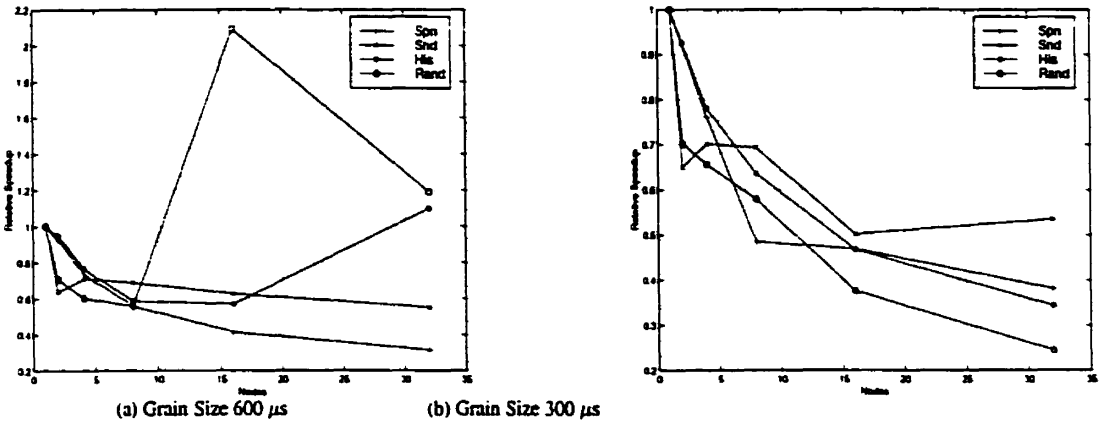
(a) Grain Size 600 μs (b) Grain Size 400 μs

Figure 5.47: Performance of SPMD(3,3,0). (a) Polling Interval 50 μs (b) Polling Interval 100 μs

In Fig. 5.47, an increase in grain size amortizes the balancer overheads of the *His* balancer so that it starts dominating the *Spn* balancer.

- For an application at a given workload, increasing grain size will not affect the relative performance of different load balancers, until a certain point when the relation between grain size, polling interval and balancer overhead favors a particular load balancer.

Fig. 5.48, part(a) shows that when the equation between grain size, polling interval, workload, and balancer overheads reaches an optimum value for a balancer, that particular



(a) Grain Size 600 μs (b) Grain Size 300 μs

Figure 5.48: Performance of SPMD(3,3,0). (a) Polling Interval 150 μs (b) Polling Interval 25 μs

balancer starts performing better. Here the *Rand* balancer is outperforms other balancers, though with increase in nodes beyond 32, its performance may degrade.

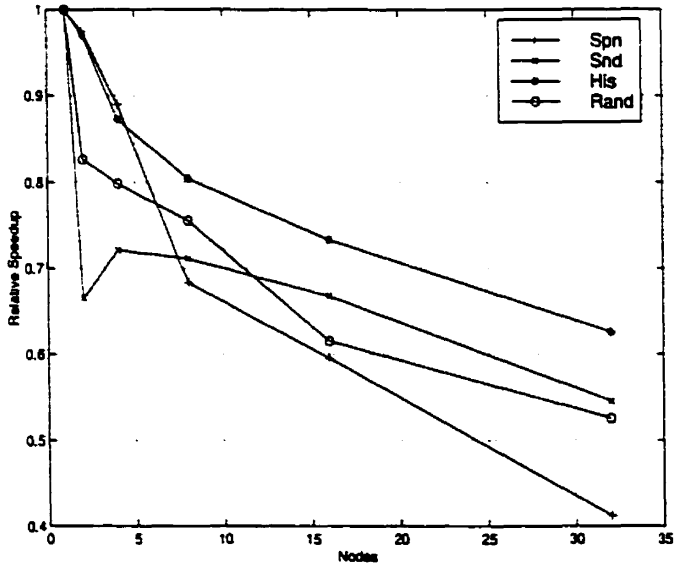
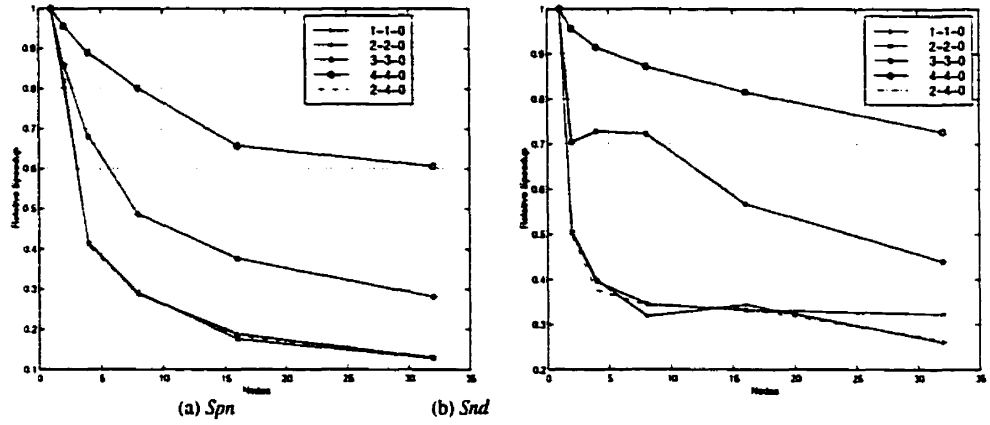


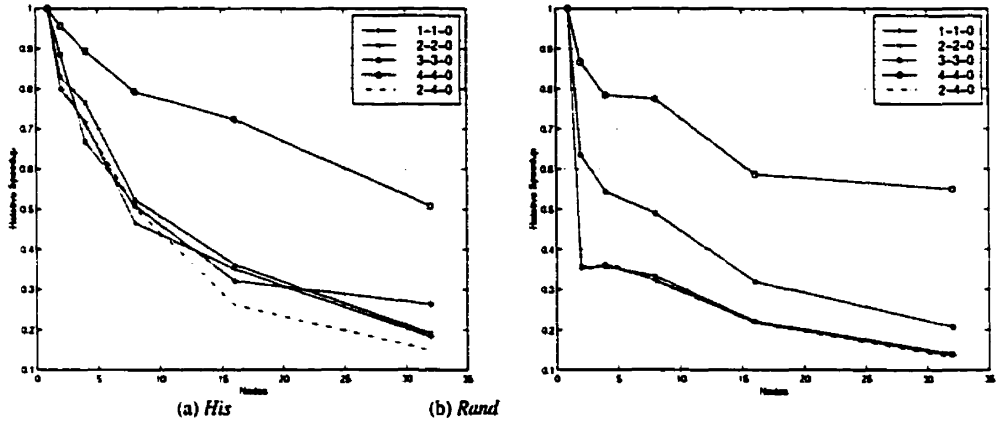
Figure 5.49: Performance of SPMD(3,3,0) at grain size of 1800 μs and polling interval of 150 μs.

In Fig. 5.49, with increase in grain size performance did not improve, as there was not much parallelism available in the workload. Therefore individual performance dips when compared to Fig. 5.48.



(a) *Spn* (b) *Snd*

Figure 5.50: Performance of SPMD at grain size 200 μs, and polling interval of 50 μs



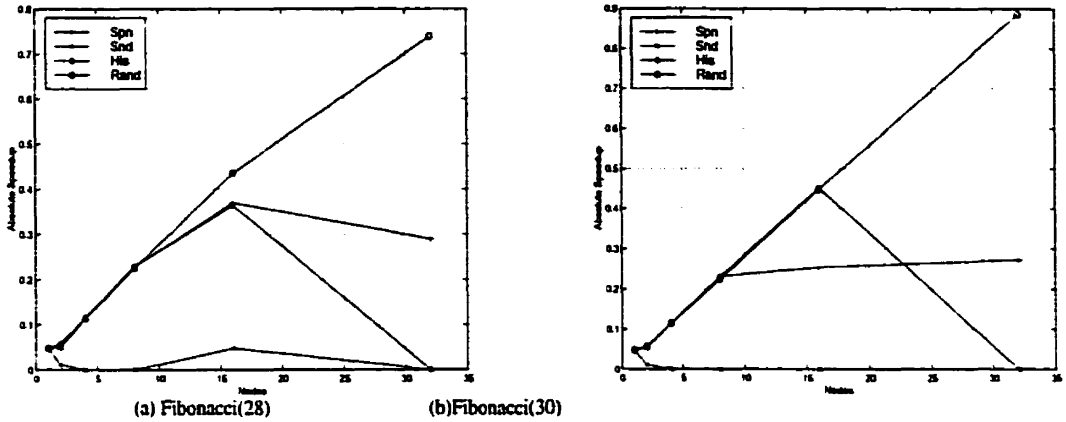
(a) *His* (b) *Rand*

Figure 5.51: Performance of SPMD at grain size 200 μs, and polling interval of 50 μs

5.5.3 Effect of Workload

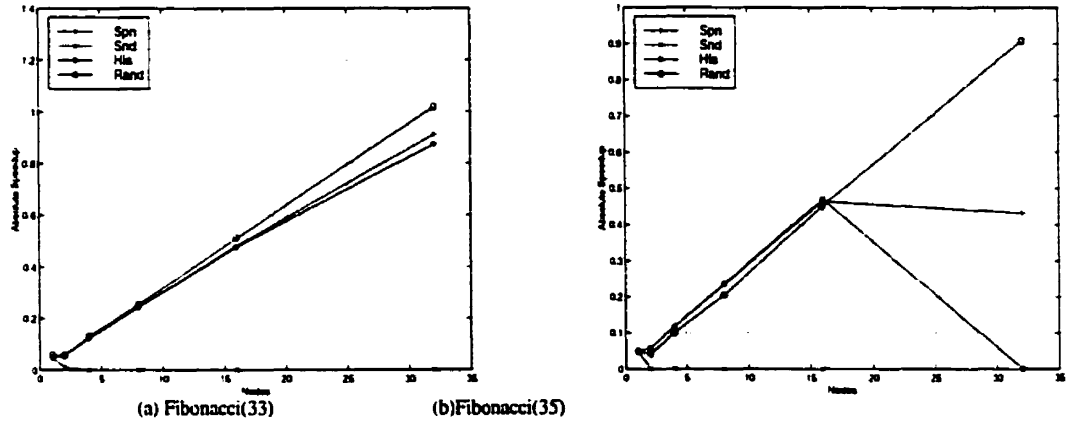
Workload is the input parameter to a program execution. The workload is the total amount of work in a sequential execution, and influences the amount of parallelism in a parallel application. Increasing the workload increases the parallelism in the application, and makes loadbalancing that much more productive. This is because the load balancer overheads are dominated by the work transferred, so a load balancer with high overheads may start performing better as the workload is increased. Similarly, a load balancer that is performing better at low loads, may be unable to scale uniformly to the parallelism in the application, and may cause instability (for instance the *Snd* balancer).

In this section, the workload is varied for the Fibonacci and the Paraffins applications, and the variance in load balancer behavior is observed. Fibonacci has extremely fine-grain threads, and the Paraffins demonstrates irregular parallelism.



(a) Fibonacci(28) (b)Fibonacci(30)

Figure 5.52: Effect of workload on different balancers for Fibonacci

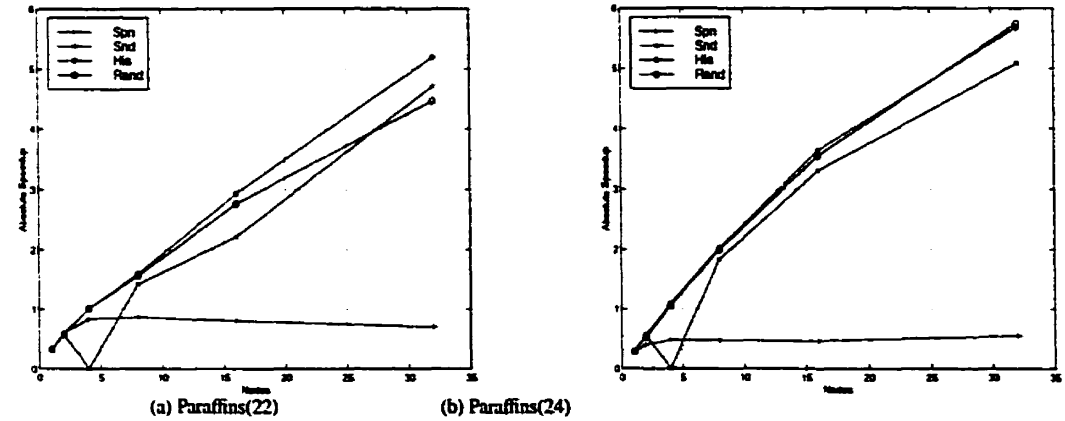


(a) Fibonacci(33) (b)Fibonacci(35)

Figure 5.53: Effect of workload on different balancers for Fibonacci

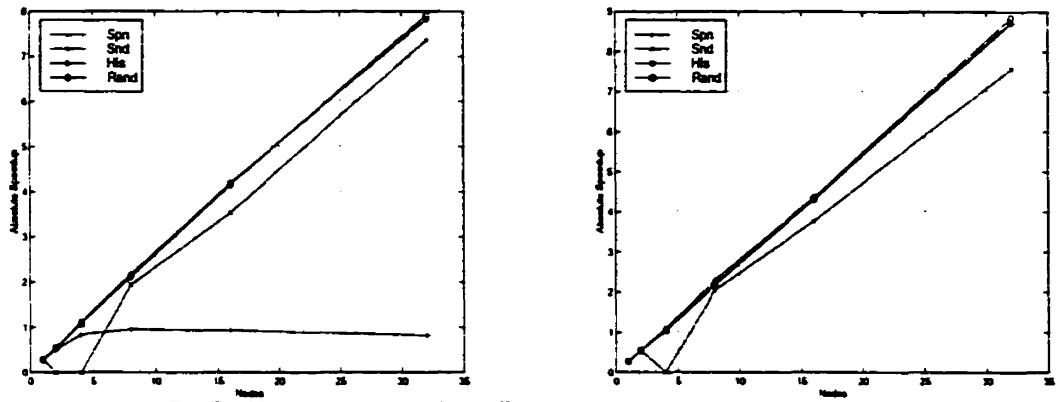
5.5.4 Effect of Application level Load Balancing

One alternative to control the amount of load balancing at the runtime system level is to determine the locality constraints explicitly in the application. The Knary(n,k,r) application specifies that r children among the total k children at every level in the n level tree



(a) Paraffins(22) (b) Paraffins(24)

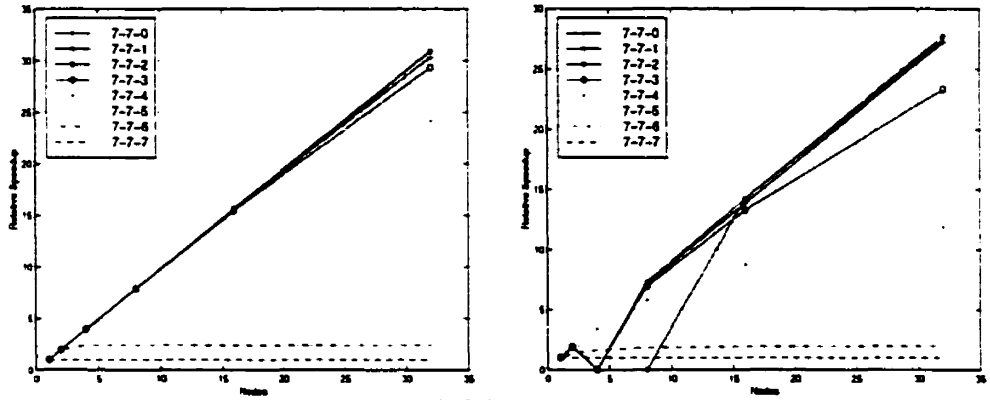
Figure 5.54: Effect of workload on different balancers for Paraffins



(a) Paraffins(26) (b) Paraffins(28)

Figure 5.55: Effect of workload on different balancers for Paraffins

should be executed locally. By controlling the value of r, different test cases for load balancers can be created.

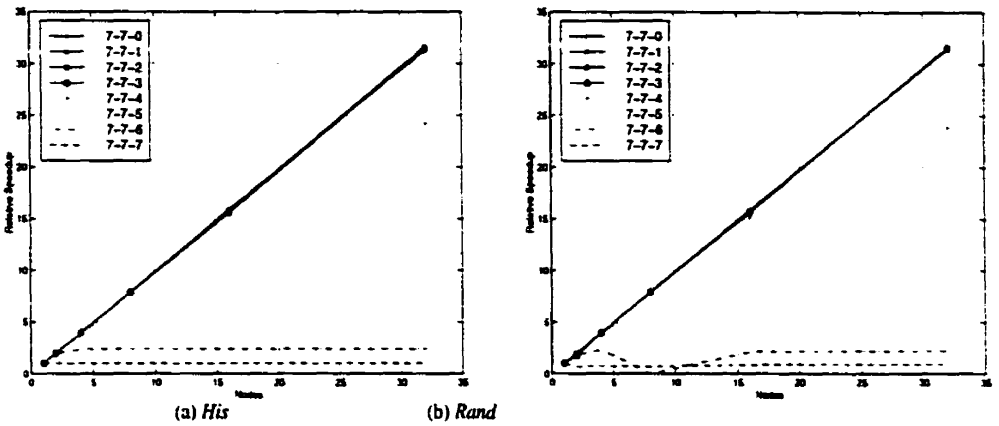


(a) *Spn* (b) *Snd*

Figure 5.56: Scalability test for Knary(7,7,X) where X is varied. X is the number of children of each node that need to be locally executed.

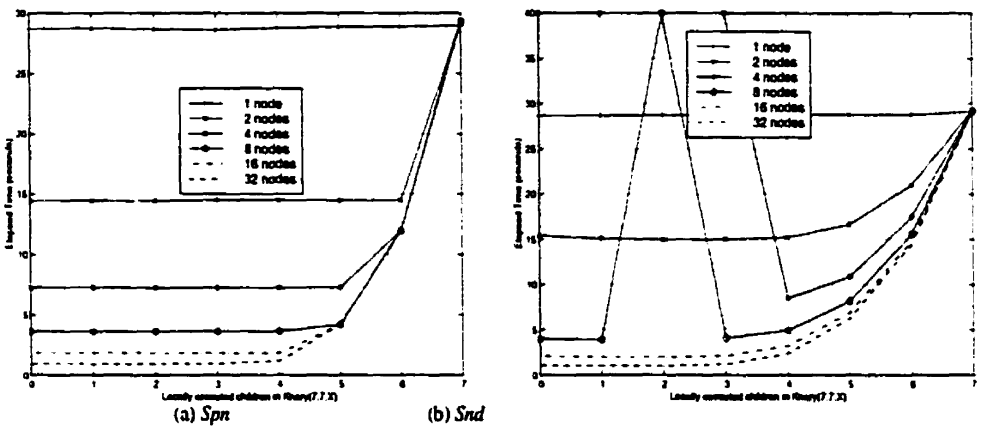
Initializing the value of X to zero allows maximum load balancing, and probably some load thrashing as well. increasing the value of X limits the amount of load balancing, and finally, the speedup becomes constant when all the children are executed locally.

Another way of looking at the effect of X on the balancer performance is shown in Fig. 5.58. here. as the value X is increased, parallel performance equals sequential performance.



(a) *His* (b) *Rand*

Figure 5.57: Scalability test for Knary(7,7,X) where X is varied. X is the number of children of each node that need to be locally executed.



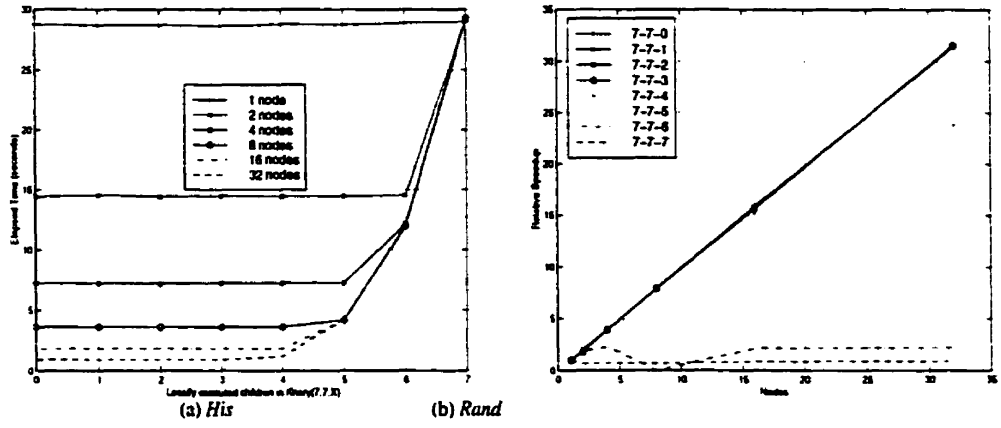
(a) *Spn* (b) *Snd*

Figure 5.58: Effect of Load balancing for Knary(7,7,X), where X is the number of children of each node that need to be locally executed.

5.5.5 Token Distribution

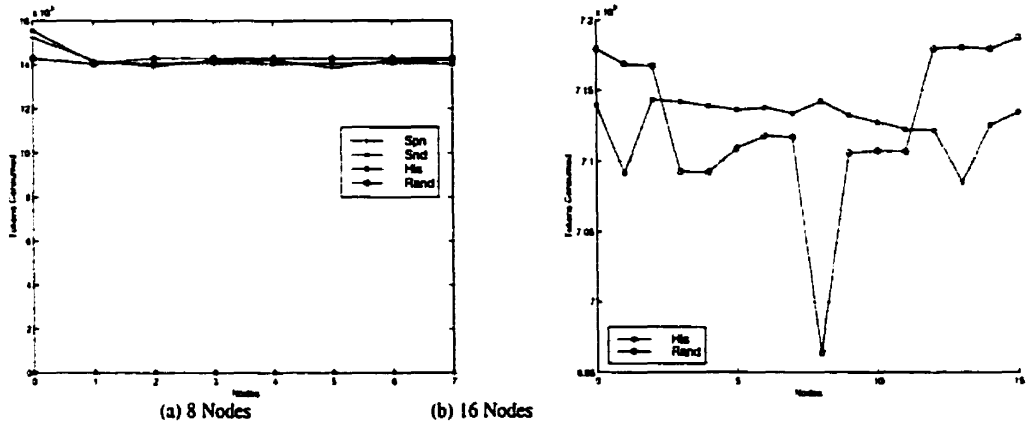
Ideally, a load balancer should distribute qork equally among all the nodes in the execution. However, with this objective, the determination of workload to be transferred is very complex process, and definetly not sustainable with fine-grain multithreading. On the other hand, the profile data reveals that balancers which distribute workload and overheads equally perform relatively better. If a line is drawn connecting all points in a graph, where each point represents the number of tokens executed on a node, then in the ideal case the line should be a straight line.

In the experiments in this section, we profiled the number of tokens executed on each node as per a particular load balancer policy, and then connect these points and observe the ability of the balancer to distribute load equally.



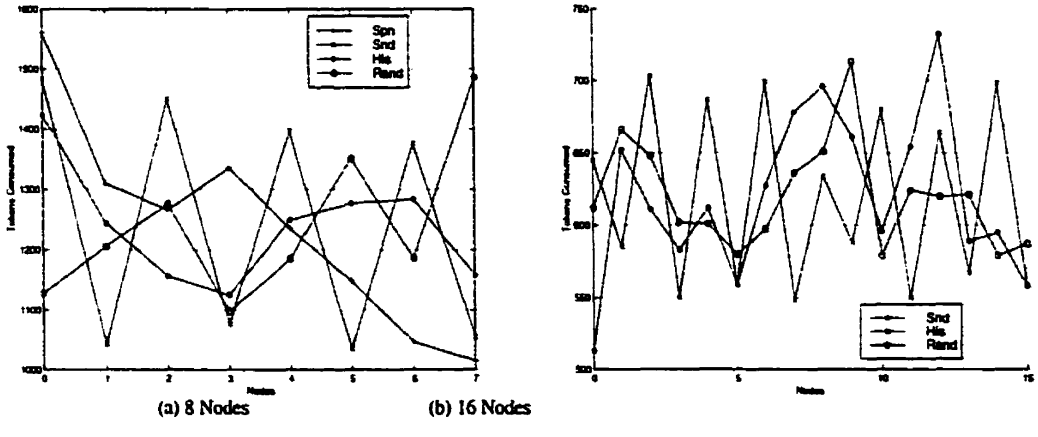
(a) *His* (b) *Rand*

Figure 5.59: Effect of Load balancing for Knary(7,7,X), where X is the number of children of each node that need to be locally executed.



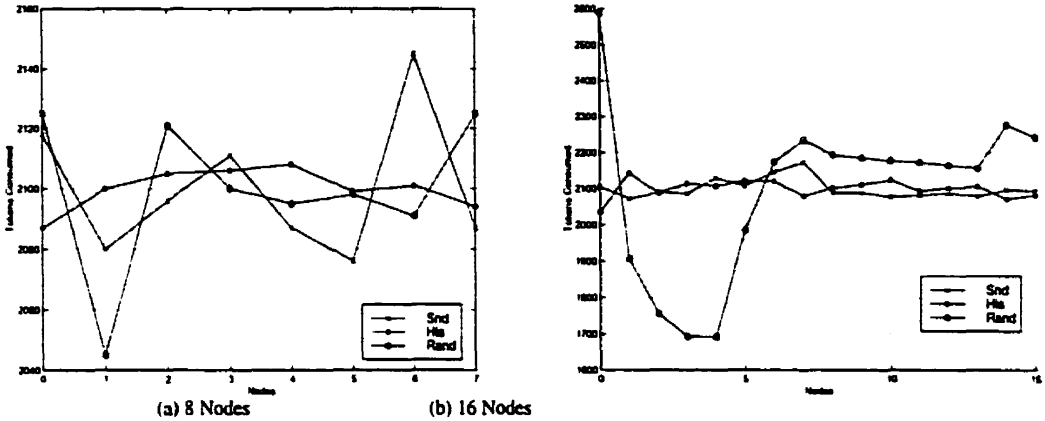
(a) 8 Nodes (b) 16 Nodes

Figure 5.60: Token Distribution for Fibonacci(33)



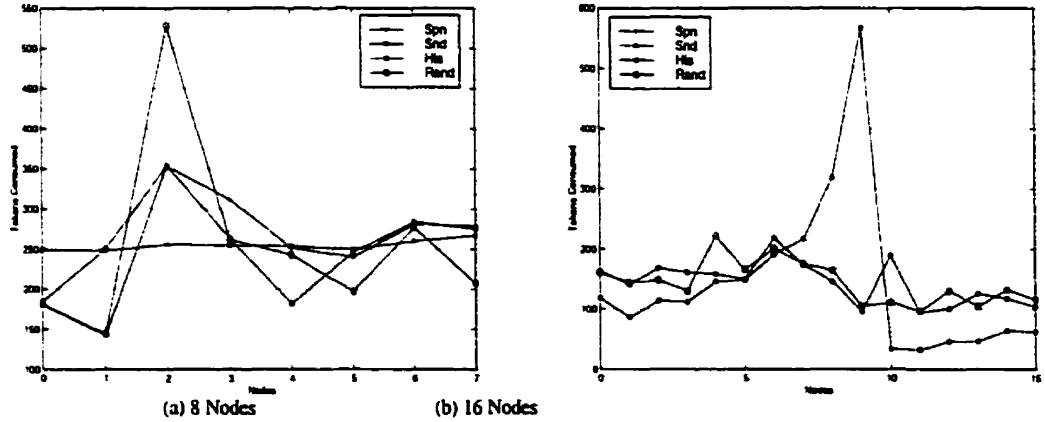
(a) 8 Nodes (b) 16 Nodes

Figure 5.61: Token Distribution for Queens(12)



(a) 8 Nodes (b) 16 Nodes

Figure 5.62: Token Distribution for SPMD(4,4,0)



(a) 8 Nodes (b) 16 Nodes

Figure 5.63: Token Distribution for Paraffins(28)

Chapter 6

EARTH Operations - A Performance Study

The overheads and latencies involved in supporting a parallel environment play a significant role in the application performance. In other words, an efficient implementation of this parallel environment with minimum costs favors lower application elapsed times. In order to understand the application performance, a detailed study of these overheads is necessary.

The parallel execution time of a Threaded-C program can be roughly divided into three major chunks: time spent in executing application threads; overheads, latencies and throughput of parallel operations; and finally the overheads involved in maintaining a multithreaded environment - load balancing overheads, initiation and termination costs of parallel execution. Application thread execution depends on various factors like the programming model, parallelism grain size, load balancer strategy, polling interval, etc. and is discussed in section 5. The load balancer overheads result from the CPU time spent in executing the load balancer code. The parallel operations in EARTH are the parallel invocations of threaded functions, thread management - instantiation, termination and synchronization of EARTH threads, and remote data communications. The Threaded-C language provides instructions to specify the EARTH operations in application programs. In this section, we study the parallel constructs in the Threaded-C language and their implementation in the EARTH runtime system from a performance perspective.

The overheads of the Threaded-C instructions are discussed in section 6.1. The latencies of EARTH operations are computed and analyzed in section 6.2. The overhead costs and throughput for data communication primitives in the Threaded-C language are

presented in section 6.3. Section 6.4 compares the throughput for local and remote block-moves of data. A similar set of experiments are performed on the other EARTH-SP platforms and the results are presented in section D.

The EARTH-SP2 at the Cornell Theory Center is considered in this section for analyzing the performance of the EARTH multithreaded environment. An important reason for choosing this SP-2 platform[1] is to maintain compatibility of the results in this section with the applications performance in section 5. This system has 137 nodes where parallel jobs can be submitted in batch mode. Each node includes a P2SC CPU running at 120 MHz, 128 KB data cache, 256 MB main memory and 256 bit memory bus. The nodes are interconnected through the SP switch [37]. The tb-3 card is the network switch interface and offers a peak hardware bandwidth of 150 MB/sec.

6.1 Overheads of Threaded-C Instructions

The overheads for executing the multithreaded constructs in a Threaded-C program are shown in Table. 6.1. Threaded-C instructions are executed on local and remote nodes and the timing costs are measured. The instruction is executed a certain `count` number of times, and the average values are presented here.

Table 6.1 shows the overheads for local and remote invocations of a Threaded-C instruction. For local operations, the time spent by a local execution unit in issuing a Threaded-C instruction is shown under the column "EU costs". For remote operations, the time spent in issuing a instruction on the source node is shown under the column "Local Costs", and the CPU time spent in executing the Threaded-C instruction on the remote node is attributed to the column "Remote Costs". Profiling code inserted before and after the Threaded-C instruction gives the total time spent in issuing and executing an instruction. Special care is taken to consider only the CPU time spent in issuing the instruction as the instruction overhead. The time spent in actually performing the corresponding EARTH operation is ignored.

When obtaining remote costs, it is important to create normal working conditions on the remote node. This is done by executing some loops on the remote node before actually issuing a remote operation. The remote operation has to compete with other work for CPU time on the remote node, before its execution. This ensures realistic overheads for remote operations. At the same time, the exact CPU time spent on the remote instruction

[1]Cornell Theory Center, Cornell University, Ithaca, New York.

EARTH Operation	Local Operation	Remote Operation	
	EU Costs	Local Costs	Remote Costs
SYNC	145.13 ns	2251.81 ns	1128.59 ns
SPAWN	109.73 ns	2043.41 ns	2058.76 ns
END_THREAD	920.30 ns	NA	NA
INCR_SYNC	168.04 ns	2205.58 ns	1207.30 ns
DATA_SYNC	170.61 ns	2173.61 ns	1149.36 ns
GET_SYNC	173.85 ns	1550.09+1971.31 ns	3448.32 ns
INVOKE(1)	119.14 ns	2177.21 ns	2339.53 ns
END_FUNCTION(1)	1044.41 ns	NA	NA
INVOKE(5)	126.74 ns	2230.15 ns	2442.57 ns
END_FUNCTION(5	1114.69 ns	NA	NA
INVOKE(9)	134.02 ns	2220.46 ns	2541.75 ns
END_FUNCTION(9)	1177.44 ns	NA	NA
INVOKE(18)	180.26 ns	2331.21 ns	2655.50 ns
END_FUNCTION(18)	1203.32 ns	NA	NA

Table 6.1: Overhead for Threaded-C instructions on EARTH-SP2

is measured by isolating it from the CPU time spent on the loops execution.

Normal working conditions are maintained on remote nodes by executing a simple `while` loop that is terminated only after completing execution of the Threaded-C instruction. In order to allow for the execution of a thread containing the remote Threaded-C instruction, the `while` loop is split into two threads as described in section 2.2.12. The thread with the Threaded-C instruction under consideration, is placed in the ready queue along with two threads corresponding to the `while` loop. This creates an environment similar to those found in normal application programs. While computing the remote costs of remote operations, the time spent on executing Threaded-C instructions is dealt with separately from the time spent on executing the loops. This is easy, as the remote Threaded-C instruction and loops are the only computation load belonging to the application on the remote node. The time spent on loops is computed by determining the time spent on executing each loop and the total number of loops executed at the end of execution.

The SYNC construct is translated into a runtime system function call to `etc_sync`. If the sync slot is present on the local node, the sync count is decremented and if its value is zero, the associated thread is placed in the ready queue. On the other hand, if the sync slot is present on a remote node, a message is composed and sent to the remote node. A

count number of synchronization operations are issued and their completion is marked by the firing of the associated thread. The timer is set off in the first statement of the newly enabled thread. The overhead of the SYNC instruction is computed by deleting a time approximation for the EARTH operation from the total elapsed time.

The SPAWN construct is less expensive than the SYNC construct, as checking for sync count is unnecessary. The associated thread is directly placed in the ready queue. The END_THREAD primitive indicates the end of the current thread, and therefore preprocessed into a `return` statement at the end of the C function representing the thread. The END_THREAD primitive pertains to the current thread and is more of a directive rather than as an instruction. Computing remote costs for it is not possible, as it is not an EARTH operation. The cost of END_THREAD is the same as that of terminating a C function.

The INCR_SYNC instruction costs more than the SYNC instruction because of an incrementing operation done prior to normal synchronization operation. A DATA_SYNC instruction places the data at destination location and sends a sync message to the associated sync slot. Hence it costs more than a sync operation. A GET_SYNC instruction determines the source node of the data from its arguments, and composes a message to the node hosting that global memory location. The receiving node then composes a DATA_SYNC message with the required data to the requesting node. Understandably, it costs more than a DATA_SYNC operation.

The INVOKE instruction launches the Threaded-C function directly on the node specified. Before making the corresponding RTS call to `etc_invoke`, the arguments for the Threaded-C function are converted into global pointers where required, and this data is placed in the runtime data structures. The variable parameter passing mechanism, as explained in section 2.2.9, is used here. On the node where the threaded function is scheduled, the parameters have to be down-loaded from the parameter pointer into their respective slots in the frame pointer. Thus it is interesting to see the overheads involved in having different number of arguments to the INVOKE instruction. The timing costs for 1, 5, 9, and 18 arguments for the INVOKE instruction are studied. As expected, they show an increasing trend as the number of arguments increases. As is the case with all Threaded-C instructions, the overhead costs for the INVOKE instruction are isolated from the latencies of parallel function invocation in the EARTH system.

The END_FUNCTION instruction indicates the end of a Threaded-C function (similar

to the END_THREAD primitive which signifies the end of a thread). The dynamic memory allocated for the activation frame is returned, as shown in the preprocessed code in section 2.2.10. To simulate common application scenarios, Threaded-C functions with `1`, `5`, `9`, and `18` parameters having just a single instruction - `END_FUNCTION` are invoked, and the results are observed. The costs represent the time for obtaining the parameters from the parameter frame, returning memory to heap, and terminating the C function representing the last thread in the Threaded-C function.

6.2 Latencies of EARTH Operations

The timing requirements for various EARTH operations like thread spawning, thread synchronization, parallel function invocation and data communication are studied in Table 6.2. The EARTH operation is initiated in one thread (start-thread) and the completion of the operation marks the firing of another thread (end-thread). All these EARTH operations end with a synchronization signal.

The time taken for the entire EARTH operation is studied with respect to the manner in which it is issued - *sequential* and *pipelined*. In the sequential issue, a single EARTH operation is initiated and completed before the next issue. In the pipelined issue, multiple instances of an EARTH operation are started without waiting for the earlier instantiations to complete. The pipelined issue is expected to result in improvements in latencies because of three reasons: the start-thread need not wait for acknowledgment from the end-thread before issuing the next instance; there will be fewer context-switches between the start-thread and end-thread; and finally, the synchronization cost to fire the end-thread is minimal in the case of the pipelined issue when compared to the sequential issue[2]. Benefits due to cache reuse, though minor in comparison, also add to the performance improvement. Both the sequential and pipelined types of initiating EARTH operations are programmed in Threaded-C, and any speedup observed is over and above the instruction-level parallelism offered by the underlying architecture.

Table 6.2 shows the sequential and pipelined latencies for local and remote operations. One uniform trend visible among all operations is that the sequential execution costs far exceed those of the pipelined execution. The latencies include the time from the issue of the operation through a Threaded-C instruction, till the completion of the operation in the

[2]The runtime system code in `etc_sync` function that is executed to place an enabled thread in the ready queue needs to be executed only once for all instances of the EARTH operation.

end-thread. This is different from the timing overheads seen in Table 6.1, where the time is strictly the overhead for executing a Threaded-C instruction.

The local sequential execution has higher costs than the local pipelined costs, because of the time savings available when certain part of the RTS code is executed only once for all instances of the EARTH operation in the pipelined execution. For example for the SYNC operation, the RTS code that resets the sync count and places the thread in the ready queue is executed only for the last instance in the pipelined execution. In contrast, this is done for every instance in the sequential execution.

Writing a word takes (DATA_SYNC_x) lesser time than reading a word (GET_SYNC_x). This same behavior is visible even in Table 6.2.

The function call operations show increasing time overheads as the number of parameters increases. This is understandable as the time spent in executing extra RTS code for uploading/down-loading each parameter to/from the parameter frame.

Operation	Local Seq.	Remote Seq.	Local Pipe.	Remote Pipe.
Sync Thread:	1117.18 ns	22750.51 ns	200.53 ns	3580.52 ns
Spawn Thread:	1094.680 ns	22623.864 ns	NA	NA
Read Word:	1263.690 ns	44695.379 ns	268.174 ns	5790.639 ns
Write Word:	1176.486 ns	44714.817 ns	225.756 ns	5751.444 ns
Fun. Call (1):	2373.947 ns	45444.014 ns	1420.094 ns	6635.911 ns
Fun. Call (5):	2464.970 ns	46504.802 ns	1505.723 ns	7111.021 ns
Fun. Call (9):	2547.251 ns	46312.331 ns	1592.606 ns	7383.772 ns
Fun. Call (18):	2641.083 ns	46734.277 ns	1678.828 ns	7016.261 ns

Table 6.2: Latencies for EARTH operations on EARTH-SP2

6.3 Data Communication

The EARTH runtime system supports global memory access over a distributed memory platform. Therefore all the remote memory access is performed through message-passing. Another reason for remote communications is dynamic load balancing. While the balancer stresses on ensuring locality between function invocations, it is also possible that varying load situations result in threaded functions sharing synchronization dependences being scheduled on different nodes. Therefore, it is interesting to note the time costs in moving data among local/remote destinations.

The overhead costs and throughput achieved by using the GET_SYNC_x and DATA_SYNC_x operations are listed in Table 6.3. The sending/receiving of data is performed to/from local and remote destinations. The entries in Table 6.3 show the overhead in nanosecs, and the throughput in MB/secs as a result of local/remote data transfers. Data of all sizes are transferred - `byte`, `short`, `long`, `double`, of sizes 1, 2, 4, and 8 bytes respectively. The difference between the start and end times of GET_SYNC_x/DATA_SYNC_x operations is the time overhead, and the throughput achieved is the total amount of data transferred divided by the elapsed time.

Initially, the source array of data is initialized on node 0 by a threaded function (say Th. A). Another threaded function (say Th. B) creates dynamic memory for the destination array and resets its contents to zero. For initiating local data transfer, the second threaded function (Th. B) is invoked on node 0, whereas for remote operations it is invoked on node 1. All data transfer happens between the source array (declared in Th. A), and the destination array (declared in Th. B).

Operation	Local		Remote	
	Overhead	Throughput	Overhead	Throughput
DATA_SYNC_B	282.00 ns/op	3.55 MB/s	5933.00 ns/op	0.17 MB/s
DATA_SYNC_S	272.00 ns/op	7.35 MB/s	5882.00 ns/op	0.34 MB/s
DATA_SYNC_L	256.00 ns/op	15.61 MB/s	5844.00 ns/op	0.68 MB/s
DATA_SYNC_D	238.00 ns/op	33.59 MB/s	5668.00 ns/op	1.41 MB/s
GET_SYNC_B	328.00 ns/op	3.05 MB/s	5921.00 ns/op	0.17 MB/s
GET_SYNC_S	322.00 ns/op	6.20 MB/s	5892.00 ns/op	0.34 MB/s
GET_SYNC_L	307.00 ns/op	13.03 MB/s	5900.00 ns/op	0.68 MB/s
GET_SYNC_D	333.00 ns/op	23.99 MB/s	5677.00 ns/op	1.41 MB/s

Table 6.3: Overhead costs and Throughput for Data Communication in EARTH-SP2

Remote memory access is costlier than local memory access (approx. 20 times). The overhead costs are almost the same for data of all sizes (though they show a negligible decreasing trend from byte to double for the DATA_SYNC_x operation). The throughput goes on increasing from byte to double, as the amount of data transferred is increasing while the time taken is almost same.

6.4 Blockmove Operations

Performancewise, moving blocks of data is more beneficial than multiple datum transfers due to low synchronization costs. A single synchronization signal is needed to signal the completion of a blockmove operation, whereas individual data transfers require as many synchronization signals as the number of data transfer operations.

The throughput resulting from block movement of data to local/remote destinations is shown in Table 6.4. Typically, block moves of data result in a throughput of around 243MB/s (local) and 95.06MB/s (remote) for 0 byte aligned data block movement.

Align	Local		Remote	
	Single	Dual	Single	Dual
0	243.22 MB/s	247.20 MB/s	95.06 MB/s	99.89 MB/s
16	247.77 MB/s	247.95 MB/s	94.28 MB/s	98.93 MB/s
8	247.13 MB/s	247.80 MB/s	91.94 MB/s	94.10 MB/s
4	238.73 MB/s	247.98 MB/s	90.88 MB/s	99.95 MB/s
1	231.24 MB/s	231.07 MB/s	87.32 MB/s	99.41 MB/s

Table 6.4: Throughput for Blockmove operations on EARTH-SP2

The results are obtained by varying destinations among local/remote, single/dual blocks to transfer, and the byte alignment of data. In the single block transfer, data is transferred between one pair of source-destination during the time observed. With double block transfer, data is transferred between two pairs of source-destinations. In either cases, a single block-move operation is performed. If 20 sync signals are to be generated to enable a consumer thread to act on the blocks of data transferred, the 20 signals may be generated after transferring single block data 20 times, or they may be split up between two block transfers (each generating 10 sync signals). In addition, the data is `0, 1, 4, 8, 16` bytes aligned. For example, the first entry in Table 6.4 shows the transfer of zero aligned data, i.e. the data from the very first byte of array at source location is transferred. Both the source and destination locations are on the local memory for local transfers, whereas in the case of remote block transfer remote data is transferred to local destination. Elapsed times are measured while transferring single and dual blocks of data. Thus the entries in Table 6.4 show throughput in MB/s achieved by moving appropriately aligned data, stored in single/dual blocks among local source/destinations and remote source to local destinations.

Local block transfer, as expected, achieves better throughput (approx. 2.6 times) than remote block transfer. This is understandable as the overhead costs associated with remote memory write and sync operations are quite high. For any alignment, the dual block transfer achieves higher bandwidth than the corresponding single block transfer. This suggests lesser time required for dual block transfer than single block transfer.

Chapter 7

A Comparative Performance Study of Fine-Grain Multi-threading on Distributed Memory Machines

This section provides a comparative study of the implementation of the Efficient Architecture for Running THreads (EARTH) on IBM SP-2, Beowulf, and the MANNA machine [93, 94]. Each platform presents different constraints on the interaction between the EARTH runtime system and the network. Threaded-C, the programming language for EARTH, provides a uniform address space to allow data exchange among the processing nodes in all these distributed-memory platforms. The performance in each implementation is characterized by measuring the cost of EARTH operations, such as the exchange of synchronization signals, the spawning of threads, and the movement of data across processing nodes. This is followed by a detailed study of the performance of applications belonging to three different programming models.

7.1 Execution Model versus Architecture Performance

Designing multiprocessor systems that deliver a reasonable price-performance ratio using off-the-shelf processor and compiler technologies is a major challenge. While modern processors can issue multiple instructions per cycle, they lack the features required

to address fundamental issues in multiprocessing systems: latency, bandwidth and synchronization overheads. A well designed parallel system must balance the trade-off between a fine task granularity [143] and the impact of communication latencies on performance. Coarse-grain parallel systems can tolerate long latencies if the application provides enough parallelism because each task is long enough to amortize the communication overheads. But coarse grain systems do not fully exploit the parallelism existing in irregular parallelism. Fine-grain parallelism, on the other hand, enables further parallelization of many applications, but has proved to be difficult to support due to the higher relative cost of communication and synchronization latencies [143].

We present performance results from three implementations of EARTH: EARTH-SP2, EARTH-Beowulf, and EARTH-MANNA. All these implementations run the same application program written in or compiled to Threaded-C, an explicitly multi-threaded extension of C. In all three implementations the Threaded-C code is converted by a preprocessor into ANSI-C with calls to runtime system functions. The translation sequence of Threaded-C programs into final executable is shown in Fig. 2.1. The runtime system performs thread scheduling, context switching between threads, inter-node communication, inter-thread synchronization, global memory management, and dynamic load balancing.

Given the EARTH programming and execution model, and its implementation on platforms with different processor-network, processor-memory and network-memory interfaces, it is interesting to study if the EARTH multithreading model can effectively deliver performance improvements for a range of applications across these platforms. One should expect that obtaining performance improvements on tightly coupled architectures should be easier than on loosely coupled ones.

7.2 Hardware Platforms

We select three machines for this comparative study: the MANNA, the IBM-SP2, and the EARTH-Beowulf. This machines represent different levels of availability, cost, and effort to implement a parallel system. The MANNA is a research machine with dual processor nodes interconnected through a cross-bar switch. The EARTH team had direct access to the network interface and hardware storage in the machine, and thus was able to produce a very efficient implementation of the EARTH model. Only a few installations of MANNA exist. The IBM SP-2 is an inherently parallel machine that is typically available in computing centers. The EARTH team was also granted access to the network

card data structures in the IBM-SP2 to enable the EARTH runtime system to directly start network operations. The Beowulf implementation uses exclusively off the shelf components, hardware, network drivers, and operating system. It is the most portable version of EARTH, and the most available because the cost and effort to construct a Beowulf cluster is minimal. However this portability imposes a hit on the latency of the EARTH operations.

The MANNA (Massively parallel Architecture for Non-numerical and Numerical Applications) was developed at GMD-FIRST in Berlin, Germany, in the early 90's [31]. Each node of the machine contains two 50-MHz Intel i860XP RISC processors, each with on-chip data cache and instruction cache of 16KB each. The two processors share 32 MB of DRAM on a common bus, and stay coherent with this memory and each other using bus snooping and the MESI protocol. The bus also runs at 50 MHz. Multiple dual processor nodes of the MANNA are connected through a custom-designed 16 $\times$ 16 packet-switched crossbars. Each input port can accept one data byte per 20 ns cycle, and the input is buffered by a FIFO. The crossbar bandwidth is 800 MB/s if all 16 inputs are in use and each transmits to a different output port. The EARTH-MANNA implementation has been described previously [156].

The IBM RS/6000 Scalable POWER Parallel System (SP-2) is a distributed memory multiprocessor. Each processing node is equipped with a 120 MHz POWER2 Super Chip, 128 KB of data cache, 32 KB of instruction cache, at least 64 MB of RAM, and operate with a 256 bit memory bus. The tb-3 switch provides a network interface with a peak hardware bandwidth of 150 MB/s in each direction. A detailed description of the EARTH-SP2 implementation is explained in chapter 2 and also in [92].

The Beowulf cluster [141] is equipped with 200MHz Pentium Pros, each node with 128 MB of RAM. The nodes are interconnected through a 100 Mb/s switched ethernet network. The EARTH inter-node communication and synchronizations are implemented on top of the TCP/IP protocol.

7.3 Latency of EARTH Operations

The machines that we are studying have different processor and network speeds, and distinct implementations of the EARTH runtime system. The latency of the operations required to communicate and synchronize across processing nodes is a determinant factor in the performance of some applications. In this section we measure the latency of some

Machine	Operation	Sequential		Pipelined	
		Local	Remote	Local	Remote
MANNA 1 cycle = 20 ns	Sync Thread	116	199	42.0	49.7
	Spawn Thread	113	213	—	—
	Get_Sync	141	348	56.8	94.0
	Data_Sync	138	333	53.0	90.7
	Fun. Call (1)	250	451	159	140
	Fun. Call (18)	410	628	276	223
SP2 1 cycle = 8.3 ns	Sync Thread	104	2751	24	414
	Spawn Thread	101	2652	—	—
	Get_Sync	122	5366	32.2	699
	Data_Sync	107	5276	27.2	695
	Fun. Call (1)	231	5553	140	784
	Fun. Call (18)	262	5656	171	831
Beowulf 1 cycle = 5.0 ns	Sync Thread	1146	21014	15.7	227552
	Spawn Thread	1193	22863	—	—
	Get_Sync	1211	41614	26.5	11482
	Data_Sync	1201	41513	27.2	37272
	Fun. Call (1)	2416	42728	1228	176389
	Fun. Call (18)	2514	43735	1339	160271

Table 7.1: Latency of EARTH operations, measured in number of cycles.

EARTH operations in all three platforms. These measurements are presented in terms of the number of processor cycles in the machine to facilitate a comparison between the machines. It is important to observe that the processor is not busy with the operation for the number of clock cycles shown in Table 7.1. Most of the remote operation time is spent either waiting on queues or in the network, releasing the processor to execute other ready threads.

Table 7.1 displays the latency of EARTH operations in the three platforms used in this comparative study. In the measurements in the "sequential" column the next EARTH operation is issued after the receipt of a synchronization signal confirming that the current operation is completed. For instance two threads, thread a and thread b, are necessary to measure the latency of a synchronization operation. Thread a issues the operation, and terminates, while the launching of Thread b marks the completion of the operation. After executing the runtime system code for the operation and thread b is enabled, thread b is placed in the ready queue for execution. In the case of the "pipelined" measurements, multiple operations are issued from thread a, which then terminates. The elapsed time is measured in thread b.

The measurements in the first row of Table 7.1 are obtained as follows:

Sequential Local: Thread a issues a synchronization signal that causes thread b to became enabled. When enabled thread b issues a synchronization signal that causes thread c to became enabled. This cycle is repeated N times (we used $N = 100000$ in our tests). The time required for the N repetitions is measured and the average per synchronization signal is computed.

Sequential Remote: Same as above but thread a and thread b are scheduled in different processors, thus there is a delay of going through the network to perform remote operations.

Pipelined Local: Thread a starts the clock and issues N synchronization signals without waiting for any synchronization signal. After receiving N signals thread b is enabled and stops the clock. This version is called "pipelined" because in a machine with separate SU and EU units, the operation of the EU, SU and the network can be superposed in a pipelined fashion. Even in single-processor nodes, this results in performance gains because the sender CPU does not need to wait for a reply from the receiver CPU, before sending the next request. In addition, the synchronization is handled in a different manner with the pipelined version, that results in fewer context-switches than in the sequential style of execution.

Pipelined Remote: Similar, but thread a and thread b execute in different processors. When enabled, thread b sends a synchronization signal to another thread in the same processor as thread a (thread c) to stop the clock.

Both in the MANNA and in the IBM-SP2 the EARTH runtime system has direct access to the network interface and can start network operations without any context switching. In fact in the case of the MANNA, the second processor performs all network related operations. In both cases, the runtime system has direct access to the network card data structures which makes network communications and polling faster. This is in contrast to the relatively high overheads associated with traversing through the TCP/IP stack in the case of the Beowulf. Further, when a message arrives, an interruption is generated to force the operating system to handle the message. This causes a context switching between the EARTH runtime system and the Linux operating system [1]. We are currently reviewing

[1]The times reported for the Beowulf runs are "wall clock time" and thus include the costs of the intervening operating system activities. This is a correct measurement because under the current implementation,

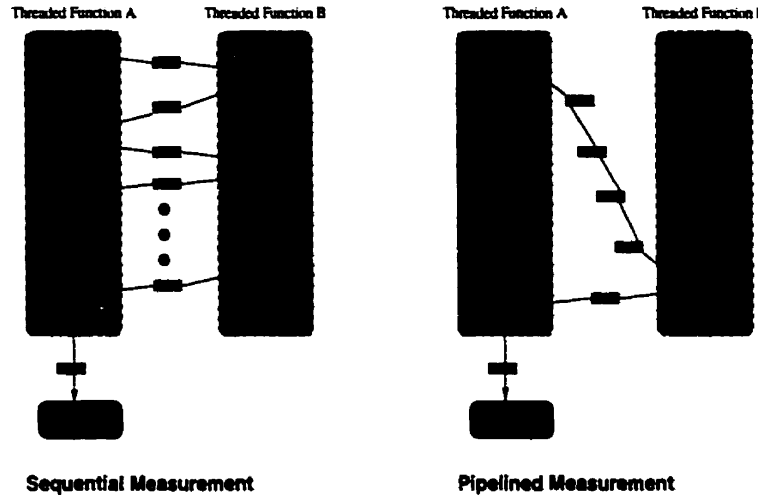


Figure 7.1: Exchange of synchronization signals for the sequential and pipelined measurements of the latency of a sync operation.

the EARTH-Beowulf implementation to reduce the penalty of the intervening OS actions in the latency of the EARTH operations.

Figure 7.1[2] illustrates the sequential and the pipeline measurements. The latency of EARTH operations are shown in Table 7.1. One observation common to most operations is the high latencies associated with sequential execution when compared to the corresponding pipelined measurements. This is expected, as the overheads associated with issuing the operations sequentially are absent in the pipelined runs. The difference in the processor speeds is very well reflected in the different pipelined speedups for the latencies for local operations (ratio of sequential latencies over pipelined latencies). This ratio is even higher in the case of the Beowulf, because of factors other than the processor speed. The EARTH runtime system polls the network at the termination of every thread. After responding to synchronization or load balancing requests, execution continues with the next thread in the ready queue. Since a sequentially issued operation is terminated in another thread, the polling costs add to the local CPU costs.

Remote operations cost less in the MANNA than in the IBM SP-2 or the Beowulf, because of the second processor in the MANNA which takes care of the communication

the user will not be able to distinguish between the time spent in the operating system and in the EARTH runtime system

[2]For the sake of clarity of presentation, a particular case is shown here. In general, the source and destination threads for EARTH operations may be in the same threaded function. Further, threaded function B can be executed either on local or remote nodes.

and synchronization operations. The remote costs for sequentially issued operations cost higher in the Beowulf, because of the time required to compose the sending and receiving messages in addition to the polling time.

Pipelined execution of remote operations on the Beowulf is an exception, where the sequential version runs far faster. This is because of the higher context-switching overheads endured between the runtime system code, and the operating system code, while sending messages across the network. After executing the runtime system code for the operation, control switches to the operating system to perform the actual communication, after which control again switches back to the runtime system code for issuing the next operation. This switch between the kernel and user space is the reason for the poor performance of remote pipelined operations.

The other EARTH operations measured in Table 7.1 include the direct spawning of a thread; a `get_sync` operation in which thread 1 requests a word of data from thread 2 and thread 2 synchronizes thread 1 when the data arrives; a `data_sync` operation in which thread 1 sends a word of data to thread 2 and thread 2 synchronizes thread 1 when the data arrives; and function calls with 1 and with 18 parameters, which represent the invocation of a threaded function either in the same node or in a remote node.

7.4 Comparison of Application Performance

In this section we present performance results for three applications: N-Queens, Paraffins(28), and a dense matrix multiply, in all three platforms. These benchmarks are described in section 4.1.

The Figures 7.2, 7.3, 7.4 show the absolute speedup for three benchmarks on each machine. The table 7.2 displays the actual execution time for the applications in the three platforms. The absolute speedup is measured as the quotient between the time required to execute a sequential version of the code and the time required to execute the parallel version in P processors.

An interesting observation to note is the disparity between the CPU speeds and network speeds. In the case of the MANNA, the slow CPU speed results in high elapsed time for sequential execution. In addition, the *Dual* load balancer provides a very simple load balancing algorithm, with minimum overheads. The extra messages generated due to the ring topology adopted in the *Dual* balancer, are compensated by a dedicated processor on each node to deal with the network traffic.

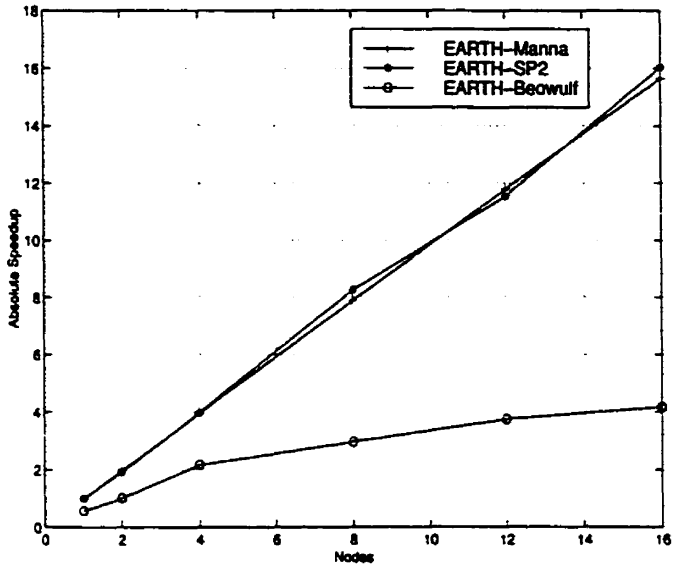


Figure 7.2: Absolute Speedup for Queens(12)

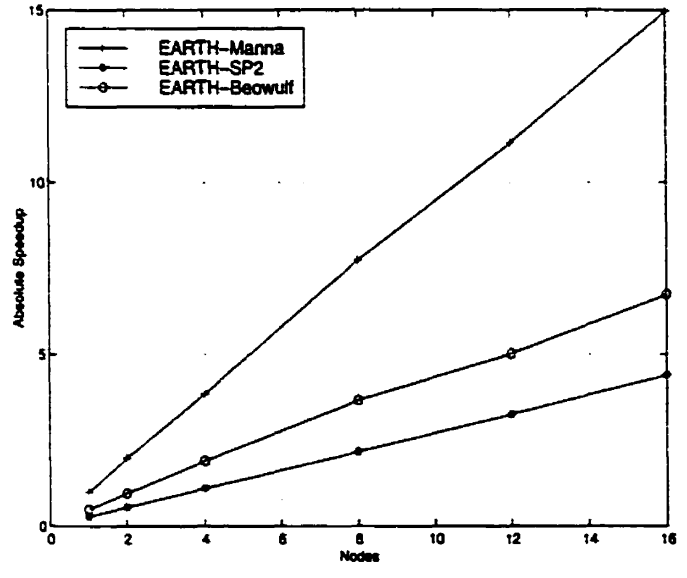


Figure 7.3: Absolute Speedup for Paraffins(28)

The *His* balancer on the SP-2 and the Beowulf, on the other hand works on single processor nodes. In order to reduce the network traffic, the *His* balancer uses history information to send tokens directly to the destination nodes, rather than following the ring topology. This balancer works very well in the case of the IBM SP-2, due to its efficient

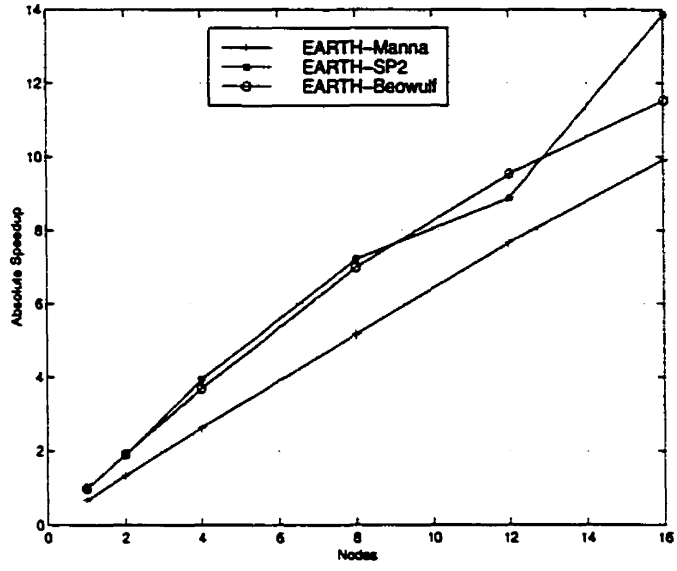


Figure 7.4: Absolute Speedup for Matrix(1024X1024)

network interface. However, in the case of the Beowulf, high CPU speed and low network speed result in comparatively poor performance, especially in the case of irregular, and communication intensive applications. Due to the high CPU speed, the computation time is usually not high enough to amortize the remote communication costs.

Another important factor is the *uni-node support efficiency* or USE factor [84, 156]. The USE factor is the ratio of sequential execution time and the elapsed time for one-node parallel execution. The USE factor is described in section 5.2.4.

In the case of Queens(12), both the MANNA and the SP-2 implementations of EARTH deliver almost linear speedup. The *His* balancer on the SP-2 performs better than the *Dual* balancer which is tuned for the dual processor MANNA. However the speedup of the Beowulf implementation tapers off after a small number of processors. We believe that this happens mostly because of the iterations between the EARTH runtime system and the Linux operating system actions, including the frequent interruptions to the kernel because of frequent arrival of small messages. In addition, the USE factor on the Beowulf for the Queens benchmark is quite low. This is because of the significant amount of multi-threaded overheads endured, despite the throttling of parallelism.

The IBM SP-2 platform performs best for the irregular application Paraffins (28). However, the elapsed time for sequential execution on the SP-2 is low, resulting in a very low USE factor. This in turn results in poor absolute speedup when compared to

Benchmark	Machine	SEQ	Parallel: Num of Processors					
			1	2	4	8	12	16
Queens(12)	MANNA	17.25	17.46	8.74	4.37	2.19	1.46	1.10
	SP2	4.79	4.78	2.50	1.21	0.58	0.41	0.30
	Beowulf	6.63	11.56	6.51	3.60	2.22	1.77	1.59
Paraffins(28)	MANNA	398	398	200	101	51.0	34.7	25.8
	SP2	57.3	206	104	52	26.4	18	13
	Beowulf	168	342	174	88.2	45.9	33.5	24.9
Matrix (1024X1024)	MANNA	364	542	271	138	70.4	36.71	30.70
	SP2	283.47	284	149.95	72	31.97	20.48	39.4
	Beowulf	245	249	128.22	66.27	34.98	25.68	21.22

Table 7.2: Execution time (in seconds) for the sequential and parallel versions of three benchmarks on the MANNA, IBM-SP2, and Beowulf platforms.

the MANNA or the Beowulf. The dynamic computation in this application is handled very well by the high speed processors of the Beowulf when compared to the MANNA. This can be observed in the low sequential execution time for the Beowulf, in contrast to MANNA.

The matrix multiplication application represents the regular class of problems, where the computation time can amortize the minimal multi-threading overheads. This is visible in the near unity USE factors for the SP-2 and the Beowulf platforms. On the other hand, with the MANNA platform, the extremely regular computation requiring lots of memory accesses fails to hide or overlap with the multi-threading overheads on the slow CPU. This application relies a lot on equal distribution of the workload by the load balancer. Here, the *His* balancer (in the SP-2 and Beowulf) scores very well against the *Dual* balancer in the MANNA runtime system. The long token distribution latencies due to the ring topology, minimal load state information of the *Dual* balancer fail to exploit the regular nature of the application and result in unequal load distribution.

The three machines studied in this paper are quite different. The runtime system is tuned to take advantage of specific features of the MANNA hardware, whereas a portable runtime system is used in the case of the SP-2 and the Beowulf. The Beowulf cluster is the most "off-the-shelf" and most affordable machine; it uses readily available processors, networks, compilers and operating systems. Although commercially available, the IBM-SP2 is not as affordable, and thus is only accessible in computer centers. Further, the runtime system has direct access to the network data structures, which facilitates lower

communication overheads when compared to the TCP/IP interface in the Beowulf. Although the MANNA is a very good platform from a computer organization stand-point, it might have a longer execution time than an SP-2 or Beowulf platform for the same number of processors. On the contrary, the Beowulf is a very affordable and interesting option for using off-the-shelf technology; however, the performance for irregular applications is limited by the network speed. The IBM SP-2 performs reasonably well for most applications, with a very effective network interface.

7.5 Performance Overview

The relatively poor performance for both the EARTH-SP2 and the EARTH-Beowulf for the paraffins benchmark reflects the difference in speed between the processor and the network of these machines. For instance, on Table 7.1 we observe that a remote sync operation on EARTH-SP2 requires 14 times more cycles than on EARTH-MANNA. On EARTH-Beowulf requires on average 106 more processor cycles to perform a remote sync operation than EARTH-MANNA.

The dense matrix multiplication algorithm used in this study was designed to test the EARTH load balancer [3]. The speedups shown in Figure 2(c) for all three machines demonstrate that the load balancer effectively distributes the processing load among the nodes.

Applications belonging to three different programming models- recursive, irregular and regular classes are studied for their performance on the three different platforms. While the CPU speed, USE factor and the load balancer adopted are seen to affect performance in a major way across all the platforms, the high communication costs associated with the network interface seemed to have a bigger impact on all communication intensive applications in the EARTH-Beowulf.

[3]Because of data locality, a blocking algorithm would deliver better performance.

Chapter 8

Related Work

Multithreaded systems are a feasible approach to exploit both regular and irregular parallelism. Today a large collection of multi-threading systems with different threaded models, and implementation platforms are available. These systems provide support for multithreading either at hardware level, with customized functional units, or at the software level, as emulators written in some high-level language. The later approach is usually preferred because of its favorable price tag, speed of development, and portability.

Dynamic load balancing is a runtime issue that has attracted a lot of attention in parallel and distributed computing. Load balancing algorithms for different applications and their impact on performance has been well documented in the work done so far. However, similar studies for multithreaded systems are still in the early stages. It is interesting to study the application of the significant knowledge gained from load balancing in distributed computing systems to multithreaded systems, especially those implementing fine-grain threads. Here, the goals and constraints for load balancing are different from those of distributed computing. The emphasis is more on minimal load balancer overheads rather than on intelligent but complicated load balancer policies. The grain size in fine-grain systems makes it imperative to strike a balance between load balancing benefits and load balancer overheads.

In this chapter we review some of the multithreaded systems focusing on their threaded models and load balancing support for irregular, data-parallel and recursive applications. Most of the multithreading systems [91] that we reviewied here are software emulations based on off-the-shelf hardware and compiler technologies. Later in the chapter we study the current and past work done in dynamic load balancing.

8.1 Threading Models

Multi-threading systems might be characterized by their threading model. Threads can be designed according to the *cooperative* multithreading model, where threads voluntarily release the CPU, or the *preemptive* model where threads can utilize the CPU only as long as certain conditions specified by the scheduler are valid. Cooperative threads can be *non-blocking* or *blocking*. In a non-blocking system, threads must run until completion. Under a blocking threading model a thread can block when an operation with long or unpredictable latency is encountered in the application. In this case the thread relinquishes the CPU, the machine state is saved for later restoration, and another thread is scheduled for execution. When the long latency operation is completed and all dependences are met, the blocked thread is rescheduled for execution. With this case, threads are blocking, and non-preemptive [41]. In a preemptive threading model, the scheduler policy which determines the running time of a thread may be based on: thread priority, time-slices, synchronization or I/O dependences, or a combination of any of these. In a preemptive threading system, threads are always blocking, and threads enter the blocked state either due to an operation in the program or due to a scheduling decision.

In a non-blocking and non-preemptive thread model, operations with long or unpredictable latencies must be executed in a *split-phase fashion*. The first phase of the operation, also referred to as the *issuing* of the operation is performed in one thread, while the second phase, sometimes referred to as the *consumption* of the result of the operation is performed in another thread. When such a thread model is chosen, a mechanism must be provided to enable the issuing thread to specify which one is the consuming thread. There is no need to preserve machine state during context-switch time.

Neither cooperative blocking thread model nor a preemptive threading model are very attractive for fine-grain multi-threading architectures because the removal of the context of a thread from the processing unit requires that the contents of the registers and the stack must be saved in a temporary user-area before context-switching, and these must be reloaded again when the suspended threads are enabled at a latter time. In addition, this model might be unyielding for the implementation of machine-independent multi-threaded platforms. Also dynamic and irregular applications might cause excessive waste of cycles when mapped to a blocking thread model.

8.2 Software Multithreaded Systems

In the classical strict data-flow model of computation, an instruction is enabled for execution when all its operands are available [66, 85, 63, 65, 68, 47, 155, 70, 77, 127, 130, 86, 123, 125, 12, 97, 133, 132, 124, 15, 17, 45, 57, 150, 140]. To enforce the enabling condition, the instructions that produce such operands must be able to send a synchronization signal to all the instructions that will consume the recently produced result. This model proved unyielding for the implementation of machines based on current standard off-the-shelf hardware and compiler technology. However many research groups have successfully implemented a model of computation that is a direct evolution of the classical data-flow model: *fine grain multi-threading*. In the later, the unit of computation is no longer an instruction, but a code-block formed by many instructions. A code-block when scheduled for execution, runs until completion without preemption or blocking due to unpredictable latencies. An instantiation of the code-block running on a processing node is called a *thread*, thus the name multi-threading for these systems. Threads, and not individual instructions, are enabled by synchronization signals. The central idea behind many multithreaded models [7, 11, 19, 27, 43, 46, 113, 114, 157, 96, 167] is to allow the execution of these threads (code-blocks) to overlap with communication and synchronization latencies.

Around the same time that architectures derived from the data-flow model were proposed, the term *thread* started to be used to refer to multiple contexts of computation in operating systems. These threads represent different lines of control that are active at the same time within an OS process. We refer to such threads as *OS-threads*. Well known OS-thread systems include POSIX Threads, Solaris Threads, OS/2 and NT Threads. OS-threads share all the resources of a process such as memory space, files, and device drivers. However, each thread has its own set of registers, and its own stack, which are either stored in heap memory (as in POSIX or Solaris threads) or in kernel space (as in NT threads). Context-switching between these threads is far easier than that between processes, as there is no need to save and restore memory pointers and other process related resources. Only the contents of the thread specific stack and register set need to be swapped at context-switch time. Programming applications at the level of these threads, rather than at the process level is advantageous because of the high-speed context-switching among threads.

There is a major historical difference between the fine grain threads discussed earlier

and the OS-threads. Fine grain threads are generated from code-blocks that grow upwards from the data-flow single instruction. A fine grain thread is the largest unit of code that can run without incurring any long latencies due to dependence on other pieces of code or on data stored remotely. OS-threads grow downward from the process abstraction in operating system. An OS-thread is the smallest segment of code that can share a set of resources with the other threads of the same process. Typically OS-threads exploit parallelism at a coarser grain than fine grain threads, and thus must execute a higher number of instructions between thread switchings.

In the multi-threading systems that we discuss here, each processing unit issues instructions from a single thread at any time [1]. An alternative multi-threading system is called *simultaneous multi-threading* (SMT). In an SMT system a single processor is capable of issuing instructions from multiple threads simultaneously [52]. Machines with such an organization use multiple threads of computation to hide the latency incurred due to the fetching of data from the local memory. An example of the later is the Tera machine [11].

Both shared and distributed memory based platforms are considered in this study. These platforms are implemented with off the shelf computers and use threads of computation to hide latencies associated with either the fetching of data from remote regions of the memory, or synchronizing among other threads. These platforms do not use multi-threading to hide the latency caused by a cache miss, i.e., as long as the memory address referenced is in the memory hierarchy of the local processing node, the reference is regarded as a local access.

Section 8.2.1 classifies existing software multithreaded systems on the basis of their implementation strategy. In section 8.3 we present an discussion of EARTH, Cilk, and TAM, three multi-threading systems with extensive effort on language support. In section 8.4 we review many multi-threading systems whose implementation is based on function libraries and that rely on OS-threads.

8.2.1 Implementations of Multithreaded Systems

All the multithreaded systems considered here are implemented in software, and are based on off-the-shelf hardware and compiler technology. These systems can be broadly divided in two classes.

[1] When these systems are implemented on top of super-scalar/super-pipelined processors multiple instructions belonging to the same thread can be issued at one time.

Language-Based Systems: These systems often offer a language with multi-threaded constructs, and a source-to-source translator to convert this language to a standard and broadly supported language, such as C. Threaded program execution is based on the support of a custom runtime system. The runtime system implements an interface with the hardware and the system level software in the machine and provides a standard interface for portable implementations of the multithreading program environment. The language offers high amount of expressiveness and flexibility ins designing multithreaded programs. Another advantage of these systems is that threads are usually non-blocking and execute in user space. Thus overheads associated with thread switching are reduced, resulting in very light-weight threads. These systems can be implemented efficiently in both shared and distributed memory platforms. Examples of systems in this class include EARTH [84, 111, 82, 150, 92, 74], Cilk [60], TAM [43], and C+- [29].

Java is a programming language [80] with support for user-defined threads. Java programs are translated into byte-codes, which is the instruction set for an abstract computer - the Java Virtual Machine. Currently, the JVM is implemented in software, and provides the runtime environment for the execution of Java programs. Java threads are blocking in nature. Early versions of the Java Virtual machine were designed to run on single processor nodes. However, with the current popularity of SMP systems, the Virtual machine for Java 1.2 maps the Java Threads API onto threads library supported by the underlying operating system.

Library-Based Systems: These systems provide a library of multi-threaded primitives to manage user level threads on top of OS threads. In this approach the management of threads requires a few system calls, which is costly in terms of execution cycles. Most of the thread library packages that we found in the literature are designed for shared memory or distributed shared memory systems. One exception is the Chant library [114] that extends the POSIX standard for light-weight threads with functionality for distributed memory environments. Examples of systems based on library of primitives include Nano-threads [19], Ariadne [113], Opus [114], Structure Thread Library [157], and Active Threads [167].

The multithreaded program is written in an existing high-level language such as C, along with some keywords that provide multithreaded functionality. The keywords

represent function names, which are defined in the multithreaded library. The function names are declared with their interfaces in the header files that are included in the multithreaded programs. The application is compiled to object code, and linked with the multithreaded library.

The basic differences in thread modeling between multithreaded languages, and multithreaded libraries[2] are as follows:

- Threads in multithreaded languages are designed bottom-up - from a few instructions, to small functions. The idea here is to clearly overlap communication/synchronization latencies with computation. Threads with multithreaded libraries are designed with the intention to reduce switching overheads between processes, by mapping parallel segments of the application into different threads. The incentive and emphasis here is more in reducing the I/O and process switching overheads, and increasing throughput and processor utilization. To summarize, while threads with languages can grow from a few instructions upwards, threads from library implementations grow downwards from the whole program down to a few routines. Language-based threads originate from the data-flow paradigm, whereas library-based threads offer performance improvements over multitasking processes.

- Threads designed with multithreaded languages are associated or synonymous with the application code, or the associated problem that multithreading is expected to solve. i.e. there is always a "code segment (+ some data)" associated with a thread. But this definition of thread does not hold for library implementations [104, 23]. Here, a thread is just a vehicle for implementing concurrency, more like a virtual processor. It is not associated with any code or data. It can run any function, any part of the user code. The thread library schedules parallel segments of the code onto a thread.

- Library based threads are good for coarse-grain parallelism. However, their relatively high overheads make them unsuitable for fine-grain multithreading. On the other hand, language based threads are a natural fit for fine-grain parallelism, as their small thread sizes allow better exploitation of parallelism in the application.

[2]Though Java offers multithreaded constructs in its language, its threads are modeled on the basis of library threads.

- Library based threads are most suited for SMP processing, rather than distributed memory. Its very difficult (almost impossible) to get the same performance as language based threads in a distributed memory environment.

- Language threads provide the flexibility, and expressiveness to design multithreaded applications. Thread design with libraries, on the other hand, is influenced more by system considerations than application semantics.

Some implementation related differences between language library based threads are as follows:

- Library based threads are usually preemptive, and therefore have to be associated with some data structure in user space to hold their book-keeping data, just as processes need to do in kernel space. This data usually consists of the thread stack, stack pointer, registers, program counter, and some thread specific data like thread id, thread scheduling priority, etc. Therefore the number of active threads at any time is given by the ratio of heap memory size, and the minimum stack size required for each thread. Where the library is implemented in kernel space, even fewer threads can be supported simultaneously. Besides imposing constraints on the amount of parallelism that can be exploited, saving system state also makes context-switching an expensive process.

- A compiler is required for translating a multithreaded language into a general purpose programming language like C. In contrast, multithreaded libraries need only to be linked to the application program.

- As the language threads are independent of the OS platform, they can run (after recompilation) on any platform, without any changes in the code that implements multithreaded functionality. On the other hand, porting library threads between different platforms depends on the compatibility between the respective OS thread interfaces.

- Collection of runtime statistics, and debugging multithreaded applications require more program involvement with library threads.

8.3 Language-Based Systems

In this section we present four fine-grain multi-threading systems - EARTH, Cilk, Threaded Abstract Machine (TAM) and Concert. Each of these systems supports non-blocking, non-preemptive threads. An exception in this category of multithreaded systems is the Java programming language. Java threads execute in user space and execution of Java programs requires a source to source translator and a runtime system just as in the other systems described here. One major point of difference though is the blocking nature of Java threads. Java threads are very useful in improving interactiveness, throughput, better resource utilization and distributed computing. However, Java threads are not very suitable for high performance parallel applications, especially fine-grain parallelism.

First we mention our own home-grown EARTH system. The development of EARTH started at the McGill University in Montreal, Canada, and continues at the University of Delaware, USA. The original inspiration for EARTH has been derived from the McGill Data-flow Machine [66]. The research around EARTH has spawned over many fields including the development of pre-processors, runtime systems, language development, application studies, source-to-source compilers, and dynamic load balancers. Recently an evolutionary path for the EARTH system was envisioned chartering the progressive development of further customized platforms [150]. The EARTH system has been implemented on the MANNA machine, IBM SP-2, Beowulf and on a SUN SMP cluster.

8.3.1 The Cilk Multi-threaded Language

Cilk is an algorithmic multi-threaded language currently designed for symmetric multiprocessors (SMP's). Central to Cilk's development is the scheduling of multi-threaded computations using a work-stealing mechanism[3]. The Cilk computation model and its implementation are described in [27]. Earlier releases of Cilk implement the memory model called "dag consistency" [28, 26]. Cilk is a succinct extension to C and has the "C elision property": when all the Cilk constructs are removed from a Cilk code, what remains is a legal C code. The most recent release of Cilk is described in [60]. The Cilk group is well known for their implementation of world-class chess programs on the Cilk platform. A unique feature of Cilk is the development of a novel debugging tool, called "Nondeterminator", that finds data races in the execution of programs [40].

The Cilk multi-threaded language processes user-level fine-grain, non-blocking

[3]Cilk threads are not mapped onto OS threads. Therefore dynamic load balancing is required.

threads in a shared memory environment. The Cilk compiler and runtime system jointly play an active role in dynamic load balancing[4]. The Cilk compiler generates two versions of target C code for each Cilk procedure - a fast clone and a slow clone. The fast clones are meant for local execution of a procedure, and the slow clones are used as units for dynamic load balancing. The Cilk runtime system [27] employs a randomizing, work-stealing scheduler and operates on a double-ended queue that is similar to the token queue in the EARTH runtime system [84]. Such queuing structure was developed earlier in the ADAM architecture [110]. While there has been theoretical study of the load balancer performance [24] there has not been much study of the load balancer with respect to overheads and alternate policies. A work-stealing, randomizing load balancer still has enough overheads to discourage load migration, especially in fine-grain multi-threaded systems. Experimental studies in EARTH have shown that a randomizing hybrid load balancer that uses load state information provides excellent performance with high scalability for irregular and divide-and-conquer classes of applications, even in the absence of any compiler support.

The generation of two clones for every Cilk procedure is an application of the *work-first* principle [60]. This principle prefers minimizing the scheduling overheads borne by the work of a computation, and specifically to move overheads out of the work and into the critical path. Work is the total time needed to execute the computation serially, and its critical-path length is its execution time on an infinite number of processors. One of the key assumptions of this principle is that in the common case, the average-parallelism of a Cilk program exceeds the number of processors in the execution by a sufficient margin. Average-parallelism is defined as the quotient of the work of the computation on one node and the time spent executing the critical-path of the computation. While this may be true for some divide-and-conquer applications, it is not a common case among applications of all classes. In addition, this assumption limits the scalability of the system as average-parallelism cannot dominate with an increase in the number of processors. Further, it is difficult to maintain good scalable speedup with a work-stealing scheduler, when compared to a hybrid balancer as in EARTH. Another interesting observation made in [60] is about the problem size. Modest to big problem sizes are required to maintain high amount of parallelism, which are required to provide acceptable performance in Cilk. It would be interesting to observe Cilk performance on applications with small to reasonable workloads which represent typical fine-grain parallelism.

[4]This is unlike EARTH, where dynamic load balancing is a purely runtime system activity.

The Cilk threading model is very amenable for the solution of divide-and-conquer problems, and is most suited for fully-strict computations [27]. While the directed-acyclic graph formed from a Cilk multi-threaded computation allows communications between parent and child procedures, it does not support communications between threads belonging to different Cilk procedures that are at the same level in the activation graph. In contrast, the EARTH threaded model enables the implementation of any arbitrary activation graph through the exchange of synchronization slot addresses.

The first release of Cilk-1 [28] was implemented on distributed memory machines. The Cilk-5 release was for the SMPs [61] with shared memory. While this version of the Cilk runtime system is released for state of the art SMP systems (8 nodes) available, it is still to be seen if there can be as many processors on SMP nodes in the near future to support massively parallel applications. The SMP version of EARTH maintains shared memory environment within each node, and makes use of the `GLOBAL` type qualifier and the existing inter-node communication layer for remote memory access.

8.3.2 The Threaded Abstract Machine

The Threaded Abstract Machine project [43] at the University of Berkeley, California presents an execution model in which the compiler controls the synchronization, scheduling and storage management. The role of the compiler in scheduling and management of threads is emphasized to take advantage of critical processor resources such as register storage and exploit considerable inter-thread locality. TAM was one of the first multi-threaded systems that were built through software emulation with minimal hardware support. The compiler translates programs written in the functional language *Id* into an intermediate language called TL0, which includes code generated for thread support [135] in a distributed memory environment. An important feature in TAM is the introduction of *inlets* which are specialized message handlers to support inter-frame communications. These inlets are generated by the compiler, one for every value to be received.

A TAM program is a collection of *code-blocks*, similar to EARTH programs which are collections of threaded functions [43]. Each code-block, like a threaded function in EARTH, consists of several threads. However, a code-block also includes code for the inlets. Since an activation frame corresponding to a code-block is allocated on a processor, all the threads belonging to a code-block execute on the same processor. For this reason, code-blocks are the units of workload rather than individual threads, as is the

case of threaded functions in EARTH. However the distribution of this workload onto the processors in the system is decided by the TAM compiler [135], whereas in EARTH the workload is dynamically distributed at runtime by the load balancer. For instance, with no support for dynamic load balancing, distributing fine-grain workload statically for irregular and dynamic applications is not trivial.

A *quantum* in TAM is the number of threads belonging to a code-block that are enabled for execution at any particular instant of time. All the threads in a quantum are executed consecutively, and values defined and used within a thread can be retained in processor registers. This is unlike EARTH, where enabled threads belonging to different threaded functions are placed in a FIFO ready queue, and therefore threads from different threaded functions execute on a first-come basis. In EARTH, threads in a threaded function usually have synchronization dependences between them. Therefore, it is highly unlikely that there many threads of the same threaded function are enabled at the same time to take advantage of TAM's register usage technique. Further, the gains from register usage as in TAM may be insignificant when there is a single or a few enabled threads in a quantum. Another difference between EARTH and TAM is the dynamic scheduling of threads. In EARTH, the ready queue (FIFO) and the token queue (DEQUE) are used for local and remote scheduling of threads, whereas complex entry and exit codes have to be generated for each quantum by the compiler in TAM.

8.3.3 The Illinois Concert C++ Language

The Concert runtime system [95, 96] proposes close coupling with the compiler and hardware to overcome overheads associated with thread management and communication in a distributed memory environment, especially when dealing with fine-grain threads for dynamic and irregular applications. The hybrid stack-heap execution mechanism overcomes multi-threading overheads, and the pull-based messaging technique minimizes communication overheads.

The Concert runtime system provides primitives for communication and thread management, as well as data-locality and load balancing mechanisms. The runtime system has the same underlying structure on both distributed memory and cache-coherent loads and stores against memory. Interaction across address spaces is via software communication and object-caching mechanisms.

The load balancing mechanism in the Concert system is geared to keep all the processors busy, balance overheads against locality considerations for irregular applications. An interesting feature of load balancing here is to allow the language specify a particular load balancer policy for a set of threads. Consequently, different load balancer policies may be chosen for a single execution. Thread placement policies include work-stealing, work-sharing, and work-sharinglocal. The work-stealing is a receiver-initiated policy, the work-sharing mechanism allows rich nodes to send extra work to randomly selected nodes, and the work-sharinglocal enhances data reuse by mapping all threads accessing the same set of objects to the same processor.

8.3.4 The Java Programming Language

The Java programming language [69, 100, 80] is an object-oriented, distributed, interpreted, architecture-neutral, multithreaded, dynamic language. Java programs are compiled into byte code, which serves as machine instructions for the Java Virtual Machine, an abstract machine. The JVM serves as the runtime system and currently is implemented in software. Java supports threads at the language level with the support of the runtime system and thread objects. The JVM assumes the responsibility for thread management.

Java threads are conceptually related to processes, but they differ from processes in that threads are user-level entities and many threads reside inside a process [23, 104]. While Java threads can be considered to be limited in features when compared to other library based threaded systems, they are preferable due to their simplicity, syntactical expressiveness in building multithreaded programs and easy to use synchronization mechanisms. One major advantage of Java threads over the library based threads is in the locking mechanism. Java makes the locking error-free by using a simplistic locking mechanism that is controlled by the JVM. Once the programmer specifies the protected sections of the code, the runtime system manages the locking of the designated areas. The Java locks are built on monitors and condition variable concepts.

The scheduling scheme for Java threads is a preemptive, priority-based and non-timeslicing algorithm that allows highest priority threads to run as long as they need to [23, 69, 103, 100, 126]. Depending on the platform, the scheduler can be time-slicing as well. The algorithm works well in user mode and makes no system calls.

The early versions of the JVM ran on single processor nodes. This gives the feel of

concurrent execution, as multiple threads compete for CPU time and are executed in overlapping fashion across the time space. At any time time only one thread is in execution. However, this does not support parallelism, as applications run faster when threads are executed simultaneously on multiple CPUs [79, 23]. The OS does not know of the existence of application threads, rather it sees the whole task in the form of a single process. It is possible to start multiple instances of the JVM on different nodes in a NOW, and communicate through message-passing. However, this is very difficult to implement and unsuitable for parallel applications not only from the efficiency point of view, but also with respect to thread modeling, and system implementation issues such as inter-node communication, thread synchronization and dynamic load balancing. This kind of parallelism is possible in EARTH because of global pointers and thread synchronization slots which are supported by active messages in the communication layer. Similarly parallelism in distributed memory is achieved in TPVM [58] by running threads within cooperating processes on different nodes, which communicate through message-passing. The *HotSpot* JVM [90] for the Java 2 platform now is multithreaded and takes advantage of the host operating system's thread model. Fully preemptive Java programming language threads are supported using the host OS thread scheduling mechanism. A major advantage of using OS threads is the built-in multiprocessing support in SMP systems and parallel execution of Java programs. Another advantage of using the OS threads is is that there is no more need for dynamic load balancing.

Multiple threads can be supported by the JVM at the same time [105]. Each JVM thread has its own pc (program counter). At any point each JVM thread is executing the code of a single method. If that method is not native, the pc register contains the address of the JVM instruction being executed. The value of the pc register is undefined, if the current method being executed is native. The local variables, partial results of a Java method are stored in a JVM *frame* for that method. A new frame is created each time a Java method is invoked and destroyed after the method terminates. Along with its own set of local variables, each frame has an operand stack[5]. At any given point of time, only one frame is active for a given thread of control. Each JVM thread has a private *Java stack*, created at the same time as the thread. A Java stack stores the JVM frames.

It is interesting to examine the JVM support for blocking threads in Java. When it is time to restore for execution a previously blocked thread after it has satisfied its synchronization requirements, the frame for the thread is made the current frame, and the stored

[5]The JVM is a stack based machine.

value of the pc register is used to restart execution inside the code for a method. This is possible because the pc register has the address of the next byte code to be executed. In EARTH, the non-preemptive, non-blocking threads are mapped into C functions, and it is not possible to maintain and access the program counter register in user space. Here, the non-blocking nature of the threads helps in avoiding this situation. Even in the JVM, the pc register is of no use when executing native code.

8.4 Library-Based Systems

In this section we present multi-threaded systems that are implemented on top of operating system based threads. Although such systems might be more portable because they can run in any machine that supports the underlying operating system, they pay a high price on the cost of system calls to implement thread switching.

8.4.1 Distributed Filaments

The distributed Filaments system [59] offer multi-threaded primitives to implement fine-grain threads in a distributed shared memory model. The Filaments runtime system implements distributed shared memory with no hardware support over distributed memory systems. The threads are blocking in nature, and favor irregular, data-parallel and recursive applications. There are multiple server threads per-node, and each server thread executes a set of sharing context filaments (called a pool). In the case of irregular and data-parallel threads, the programmer/compiler has to assign context-sharing filaments to pools on different nodes so as to maintain locality and equal task distribution. However, a simple receiver-initiated scheduler distributes workload in the case of recursive threads. This balancer queries other nodes in a round-robin fashion to steal work. A filament blocks when a long latency operation is encountered. Though there is a provision for the programmer/compiler to enable/disable load balancing in Filaments, it is difficult to estimate runtime load imbalances at compile-time, especially in the case of fine-grain applications.

8.4.2 The Opus Language

The Opus language [114] provides Fortran language extensions to support task and data parallelism. Independent tasks representing coarse-grain parallelism, communicate and

synchronize through monitor-like structures called shared-data-abstractions. The Opus runtime system relies on a light-weight threads package called Chant, to support multi-threading functionality in a distributed memory environment. The Chant threads package extends the Pthreads interface with primitives for remote communications, remote thread operations by using existing communication library (MPI standard). Workload has to be mapped onto different nodes by the programmer/compiler keeping in mind locality of the tasks as there is no runtime dynamic load balancing support.

8.4.3 TPVM

TPVM is a threads based interface to the PVM distributed computing model [58]. TPVM is built as a subsystem of PVM [144] in order to address some disadvantages of a process based model. The design goals of TPVM are minimum task initiation and scheduling costs, overlapping computation and communication on a single processor, smaller granularity, and supporting event or data driven, active message based computation [145]. While the Chant threads package extends the Pthreads interface with message-passing primitives, TPVM takes the opposite direction - it adds a threads subsystem to an already existing process based, message-passing computation model.

A *thread* in TPVM is a light-weight process as defined by the underlying threads subsystem - essentially a subroutine/procedure (or a code segment including nested procedure invocations, identified by one entry point). The first version of TPVM is based on GNU/REX threads package [42]. Units of computation are threads rather than processes. Processes only serve as shells around constituent threads. An important feature of TPVM threads is their coarse-grain data-flow model. Threads are activated for execution only after all their dependencies have been satisfied. This means that the threads are *non-blocking* in nature. The purpose of this feature is to delay the binding of work units to computational resources until they are fired for execution. This reduces workload allocation and scheduling costs in NOW, where resources are not dedicated. Another advantage is that it removes complexities in the underlying message-passing model by relieving the programmer of the burden of task creation and synchronization. In order to fire waiting threads for execution, trigger messages have to be sent, and these trigger messages are based on the active message model [163]. Remote memory allows asynchronous read and write of a thread's address space by another, even when they are part of different processes, and reside on different machines. The implementation of TPVM threads over

the REX threads package is very close to that of EARTH. The TPVM runtime system includes a master thread and the TPVM library system[6] for each process, and a thread server module. A version of the TPVM threads also runs on the Solaris threads library, where unlike the description here , the threads run in a preemptive environment.

The TPVM threads are very similar to EARTH threads. The EARTH threaded model is two-layered with threaded functions and non-preemptive threads at a finer level. Threads in EARTH are associated with sync slots, whereas TPVM threads are identified by their thread id's. Load distribution in TPVM is done at compile time, and there is no reference to dynamic load balancing TPVM threads, or how the dynamic load balancing primitives in PVM are supported at the threads level. Experimental studies with TPVM have shown that TPVM is not suited for regular SPMD style of applications, whereas EARTH with its extensive load balancing support has been successfully applied for different models of computation. PVM supports utilization of processor cycles in a heterogeneous workstations, and while this is a very positive feature, one casualty is the accuracy of execution times over CPUs of different speeds and configurations. this accuracy is very important for high performance parallel applications, and also for effective load distribution. Another factor is a reliance on a TCP/IP stack for network performance. The communication overheads might be significant for fine-grain parallelism, unless the thread model allows aggressive exploitation of parallelism in the application. Enough amount of work has to be represented in the form of abundant quantity of fine-grain threads. TPVM currently limits the number of threads to 256 at any given time for each process, though this restriction is to be lifted soon.

8.4.4 Nano-Threads

The Nano-Threads [19] are user-level threads built on top of kernel threads. The Nano-threads library provides primitives to support multi-threading efficiently in a multi-user/multiprocessor environment with shared memory. A compiler takes as input C/Fortran programs with Nano-Threads keywords, and generates target C/Fortran code (Nano-Threads) along with code to manage an intermediate representation of varying levels of parallelism in the application, called the Hierarchical Task Graph. The associated code chooses the appropriate granularity for execution at runtime, depending on the availability of resources. Each Nano-Thread is associated with a per-thread-counter

[6]TPVM library primitives depend on PVM library for services such as message-passing.

and a nano-thread descriptor. Nano-Threads block so that child threads can access local variables from the address space of the parent nano-thread. All enabled Nano-threads are placed in globally accessible and manageable ready queue called GQ (FIFO). To preserve locality, each node has its own local queue (FIFO) that is accessible from all nodes. The objective of load balancing in the Nano-Threads system is to distribute the load equally among all the nodes. This is a different goal from the one adopted on EARTH, where the aim is to keep all processors busy, thereby minimizing balancer overheads in an extremely fine-grain environment. Another potential balancing overhead may be the contention problems for controlling the global queue which may degrade scalability of the system.

8.4.5 Active Threads

The Active threads library [167] define an interface for supporting fine-grain, non-preemptive, blocking threads over traditional kernel threads. They can be used to hand code applications, or as virtual machine target for compilers of parallel languages. Threads sharing context are grouped into bundles. Each bundle has its own scheduler and the scheduler may be chosen by the application from a set of schedulers distributed with the active threads package. The scheduler maps active threads onto processor thread dispatch buffers for each processor. Though the fast threading primitives ensure low overheads for thread operations, the multi-threading overheads for thread initialization, context-switching, thread stack management and synchronization are quite high for irregular applications employing fine-grain threads. In contrast, context-switching in EARTH is as cheap as a C function call, and there is no need for thread stack management.

8.4.6 StackThreads

StackThreads [147] provide low-level support for fine-grain software multithreaded environment. on stock microprocessors. Thread management is performed by calling StackThreads primitives, which are provided as a library, and can be used as compilation target. Supporting high-level abstractions on top of these base primitives is left for language designers and implementers. Unlike other multithreading schemes, it does not assume a customized frame format designed for a particular programming language or a set of multithreading primitives. Instead, it operates on standard C stack frames and calling conventions.

The threading model is blocking in nature. When a new thread is forked, the procedure comprising that thread is called as a sequential function call. When the new thread blocks, the caller is resumed by moving the new thread's frame from the stack to the heap and unwinding the stack. When the blocked thread is rescheduled, the context is restored on top of the stack and control transfers to the point where the thread blocked earlier.

The multithreading mechanism here, does not provide for thread migration. Further, this scheme does not address location-transparent access to data.

8.4.7 Structured Threads

The work on structured threads [157] at Caltech provides multi-threading support for high performance parallel applications on top of kernel threads in Windows NT. Applications can be written at two levels: as a pragma based notation in Multithreaded C, or as library calls to the Sthreads library at a lower level. The Sthreads library is built as a very thin layer on top of the Windows NT thread interface. Multithreaded C is implemented as a source-to-source preprocessor that directly transforms annotated blocks and `for` loops into equivalent calls to the Sthreads library. The Sthreads library and Multithreaded C preprocessor are integrated with Microsoft Developer Studio Visual C++. This work is aimed at providing a structured, light-weight, and less complicated threading environment on top of OS threads in SMP systems.

Application threads are mapped onto the kernel threads. Thread scheduling depends both on the Sthreads library and the kernel threads scheduler. Therefore there is no explicit dynamic load balancer. However, in order to adapt dynamically to varying load conditions and to offset the thread management and synchronization overheads, the thread model allows dynamic creation of large numbers of lightweight threads that can take advantage of whatever processor resources become available during execution. This model is suitable only for SMP systems, and therefore scalability is limited by the number of processors in a node, and the maximum number of threads that can be profitably supported on each node.

8.4.8 DSM-Threads

Distributed Shared Memory threads [121] support distributed threads on top of POSIX Threads (Pthreads) via distributed shared memory (DSM). The goal is to support migration of applications from a concurrent programming model with shared memory

(Pthreads) to a distributed model with minimal changes of the application code. The reasons for this migration are the significant computational capabilities of network of workstations, their cost-effectiveness, and the limited scalability of SMP clusters (typically no more than 40 CPUs) due to the system bus bottleneck. A programmer may continue to use the shared-memory algorithms and exploit the processing power of distributed systems without dealing with the more complex models of distributed algorithms. The DSM runtime system is itself implemented as a multithreaded system over Pthreads on each node and copes without compiler or operating system modifications.

Distributed virtual shared memory is used to address the absence of global state in a distributed system. Address references can be distinguished between local memory accesses and DSM accesses, thus creating a NUMA architecture.

The threads interface allows the programmer to specify a destination node during thread creation. If no destination is specified, the DSM system will select such a node using a history of load information and CPU throughput. For repeated executions of an application, trace data and thread group information may be used to distribute threads upon creation.

8.4.9 Ariadne

Ariadne [113] is a user-space threads library that is modeled for process-oriented parallel and distributed simulations in multi-user environments. Ariadne threads are implemented in shared and distributed memory models. Each thread is assigned an identifier that is unique to its host process. Along with the process id, this forms a unique combination to identify a thread among a system of processes. non-blocked thread gets executed first. The built-in scheduler allocates portions of a host process's time-slice to its resident threads. The internal scheduling policy is based on priority queues, i.e. a highest priority non-blocked thread gets executed first. Within a priority class scheduling is FIFO. An executing thread continues to run until it terminates, completes a time-slice, or suspends execution. In addition, Ariadne also provides for customizable schedulers. This library is more suited for coarse-grain parallelism.

Ariadne's support for concurrent execution of threads on shared-memory multiprocessors precludes the OS kernel involvement due to portability requirements. The multiprocessing power is exploited by multiplexing threads on distinct processes, generally using as many processes as available processors. The threads interact via Ariadne primitives

which in turn operate on shared memory.

Ariadne threads form the basic unit of computation in the distributed model. Threads can move between all processes in the distributed environment. Typically, threads move to access global objects at other Ariadne processes - as computations that chase data. For thread migration, Ariadne depends on the use of an object-locator: a migrating thread needs to know which host it must migrate in order to access required data. The communication layer in Ariadne is based on any arbitrary communications subsystem, such as PVM 3.3.4 and Conch.

8.4.10 Athapascan

Athapascan-1 [36] is a data-flow language designed for parallel computation. It is implemented as a C++ library for multithreaded parallel programming. Explicit parallelism is expressed through asynchronous remote procedure calls, denoted as tasks, that communicate and are synchronized through shared memory. Application execution is data-driven: the precedences between the tasks, the needed communications or the data copies are ensured by the runtime system. The scheduling of the created tasks is enforced by customizable schedulers that are fully separated from the application.

8.5 Dynamic Load Balancing

Load balancing algorithms have been an active topic of research in the distributed computing field [168, 50, 51, 136, 73, 9, 129, 48, 39, 139, 89, 169]. Various load balancing algorithms, as well as comparative studies of their performance have been published. Often the applications considered are either too regular in nature with high coarseness, or the balancers studied are relevant only for particular architectures and network topologies on which the studies were conducted. Another important concern is the definition of fine-grain parallelism. It is well known that in fine-grain applications the CPU time spent on communication overheads dominates the computation time [142]. In the process based model of distributed computing, this definition implies a high number of processes with small grain sizes. While this model supports the exploitation of parallelism at a finer level, the grain size is still relatively higher than the grain size in typical fine-grain multithreaded applications (order of μs). Fine-grain threads allow parallelism at instruction

level [84, 60, 43] and are usually non-preemptive and non-blocking[7]. Studies on load balancing in distributed computing provide useful information, however, they cannot be directly applied to multi-threaded systems where the equation between quality of balancing decisions, balancer overheads, load imbalances, and application grain size is very delicate. This is even more important for irregular and dynamic applications where the computation and communication patterns cannot be identified at compile time.

While there has been a good understanding of load balancers behavior in distributed systems [16, 122, 21, 118, 22] , the study of dynamic load balancers for fine-grain multithreaded systems is still in the early stages. Existing studies are often purely theoretical, based on queuing models or simulations. On the other hand, the results in EARTH are based on an actual multithreaded emulator built on top of off-the-shelf processors, with real applications. In this section, we review work done in dynamic load balancers for distributed computing systems, and compare the policies with those in EARTH wherever applicable. The load balancing policies studied here, distribute tasks belonging to a single application among the nodes participating in the execution. Further, we do not consider static load balancing or thread partitioning/placing policies.

Adaptive load sharing for distributed systems is studied with respect to the relative advantages of load sharing policies with increasing levels of sophistication and global state information is documented in [50]. Three sender-initiated load sharing policies are modeled: *random*, *threshold*, and *shortest*. The random policy selects a node at random for load migration. The threshold policy polls the load state of other nodes until a node is found whose load is less than a threshold. The shortest algorithm probes a set of randomly chosen nodes for their load status, and chooses the node with the shortest queue length. Only non-executing tasks are migrated. The authors show by analytical modeling that the random policy improves performance against no load balancing, and the threshold policy performs very well with its limited system state information, and the shortest policy performs best with its global load information, though not significantly better than the threshold policy. We have implemented two load balancers in the EARTH runtime system (*Rand-Rcv* and *Rand*) that are similar to the random and shortest balancer policies. The *Rand* balancer works in a hybrid[8] mode, and uses global load state information. The *Rand-Rcv* balancer performs poorly in relative comparison with other balancers, though

[7]We define fine-grain threads as threads with very small grain size, independently synchronized units of computation, and non-blocking in nature.

[8]Also referred to as *symmetric* policy.

it does well against a no balancer situation. In contrast, the *Rand* balancer is the best balancer for different classes of applications. The reasons are obvious. Firstly, the grain size of threads in EARTH is very small compared to processes considered in [50]. Therefore, the load balancer has to be very lean, and should make intelligent decisions when using network-based communications or randomizing algorithms. Secondly, the balancers in [50] work only in sender-initiating mode, unlike the hybrid nature of the *Rand* balancer. Finally, the execution model of the EARTH system is different from that assumed here. For instance, one of the assumptions made on the application model is that, all nodes are subjected to the same average arrival rate of tasks, which are of a single type. This assumption is not valid for EARTH, where the task arrival rate, communication and computation patterns cannot be predicted. Another factor in the assumption in the analytical model that the cost of probing a node is negligible. This constitutes a significant cost especially when compared to typical fine-grain threads in EARTH, as it involves packing a message and communicating over the network.

Eager *et al.* [51] compared two policies for adaptive load sharing: receiver-initiated and sender-initiated. In both cases the victim node is chosen at random. If that choice turns out to be wrong, the probing process is simply repeated until a limit is reached. Based on a queuing model and simulation results, they conclude that the sender-initiated policy is preferable since, in their model, the receiver-initiated policy would require the migration of a running task. This assumption is not applicable to the EARTH system, where tokens[9] rather than executing tasks are allowed to migrate.

The diffusive method [166, 81] is a well explored load balancing mechanism in the distributed computing field. Each node in the network calculates how much of its workload needs to be transferred based on its local load information and a diffusion equation. Then, the system exchanges work units accordingly between the nodes. After several iterations, the system load will become balanced. One of the problems with this approach is that it usually needs a lock-step mechanism to synchronize the nodes and requires that the system load doesn't change much during the diffusion phase. It also assumes that it is possible to migrate work units. However, the EARTH load balancer is only allowed to decide where *new* tasks should be allocated; i.e. the destination nodes for new tasks. Generalized dimensional exchange (GDE) [44] and hierarchical balancing (HB) [81] are other common load balancing methods, which face the same problems in a fine-grain multi-threaded environment as the diffusive method.

[9]Tokens consist of context-sharing, non-preemptive threads.

The trade-off between knowledge - the accuracy of each load balancing decision, and overhead - the amount of added processing and communication incurred by the balancing process, is illustrated with five different dynamic load balancing schemes in [168]. The *sender (receiver) initiated diffusion* strategies are asynchronous schemes which use near-neighbor information. The *hierarchical balancing method* organizes the system into hierarchy of subsystems within which balancing is performed independently. The *gradient model* employs a gradient map of the proximities of under-loaded processors in the system to guide the migration of tasks between overloaded and under-loaded processors. The *dimension exchange method* requires a synchronization phase prior to load balancing and then balances iteratively. All five strategies have been implemented on an Intel iPSC/2 hypercube. They show that the RID approach performs well, and can most easily be scaled to support highly parallel systems. Receiver-initiated balancers perform very well in EARTH, but not as well as hybrid balancers that rely on load state and history information. The load balancing model in this work is very much different from that of the EARTH system. Unlike EARTH, load-balancing in [168] tries to balance load on all the nodes equally. Secondly, their load model assumes a fixed number of tasks present in the system at initialization time. All tasks are independent and may be executed on any processor in any sequence, unlike locality and dependency constraints in a multithreaded model. Consequently, they do not consider dynamic task creation as in any multithreaded system. Further, in order to simplify the workload characterization, each task is estimated to require equal computation time. In contrast it is not possible to estimate the grain sizes of EARTH tokens. Another important feature concerns the information policy. All the nodes periodically exchange their load information with other nodes in their *balancing domain*. Each balancing domain can extend from the neighboring nodes to all nodes in the system. Understandably, periodic load update messages result in high overheads, which are difficult to be amortize by fine-grain workloads in the EARTH system. Instead, more sophisticated load update policies are implemented such as: piggy-backing every load balancing message with load state information, using load probes on randomly selected nodes, taking advantage of the sending/receiving paths for load transfers, and using history information. Information policy in the dynamic load balancers in EARTH is a demand-driven policy unlike the periodic policy here. It is well known that demand-driven policies reflect the system state better in the load balancing decisions unlike periodic policies [136]. Care is taken to consider the aging factor of available load information.

A parabolic load balancing method for problems in computational fluid dynamics is presented in [73]. This paper presents a diffusive load balancing method with emphasis on scalability to a large number of multicomputers interconnected with mesh topology. The work here presents a parabolic load balancing method, proves its correctness, convergence and scalability, and simulates applications to generate problems in computational fluid dynamics. Randomizing algorithms though considered scalable and flexible are not considered here, because the assumption that disturbances occur frequently and have short life spans is not valid in CFD applications where disturbances arise occasionally and are long lasting. The load balancing method here is based on the properties of the parabolic heat equation, which describes the diffusion of heat energy from hot regions into cold regions until the entire volume is of the same temperature. This study applies the finite difference techniques to derive an unconditionally stable discrete form of the heat equation, and uses a scalable iterative method to invert the resulting coefficient matrix. The goal of load balancing here is to ensure that all processors have equal workloads, unlike the EARTH system where stress is on keeping the processors busy. In this algorithm, each processor concurrently executes an arithmetic iteration which calculates an expected workload at each processor. Processors periodically exchange units of work with their immediate neighbors in order to make their actual workload equal to the expected workload. This method preserves adjacency relationships among elements of a computational domain, thereby minimizing the communication costs.

An adaptive heat diffusion scheme as well as a task selection mechanism that can preserve or improve communication locality is presented in [164]. The goal of load balancing is to find a mapping of tasks to computers that results in each computer having an approximately equal amount of work. This is in contrast to load balancing strategy in EARTH, where the aim is to keep all the processors busy. All phases of a typical load balancing algorithm - load evaluation, profitability determination, work transfer vector calculation, task selection, and task migration are described. This algorithm is not applicable to systems like EARTH, where it is not possible to evaluate complex equations before taking load balancing decisions.

The multi-threaded model is applied for parallel adaptive partial differential equation solving in [41]. An interesting feature in this work is that the dynamic load balancer runs as a thread, and competes with the application threads for CPU time. In addition to application execution, multi-threading here is used as a mechanism for the concurrent execution of actions required for load balancing - information dessimination, decision

making and data migration. Application threads can be in *new*, *ready*, *running*, *blocking*, or *dead* states. At thread creation time, each thread is associated with a counter, similar to a sync count in Threaded-C. Once all the dependencies for a thread are satisfied, and the counter reaches zero value, the thread is placed in the ready pool. The *co-operative* threading model is adopted here, i.e. threads run to completion or voluntarily yield the CPU. Thread scheduling is *non-preemptive*. An advantage of this scheduling strategy is the reduction of overhead by minimizing the number of context-switches. This threading model is a perfect example of those systems in which, threads are non-preemptive and blocking. EARTH supports non-blocking, non-preemptive threads. The goal of load balancing is to minimize idle times on the nodes, rather than to balance workloads equally among all the nodes. This approach is similar to that used in EARTH. A significant deviation in the load model from other systems is the process of *h-refinement*. In this process, the mesh is refined in areas where the resolution of the solution is larger than a given tolerance. After the mesh refinement, each thread can be split into two or more threads depending on the required load balancing resolution. The load balancing strategy is receiver-initiated. An analytical model is built and the results are verified with experimental observations to show that it is beneficial to support load balancing as a thread. This feature is not viable in the EARTH system, because, our load balancing is more demand-driven, and services recursive, irregular and regular applications of very fine grain sizes. Therefore, it is important in EARTH that load balancing actions consume minimum CPU time when compared to fine-grain application threads, and be as unobtrusive as possible. Furthermore, there is no concept of priority for threads in the EARTH model. Therefore, we cannot guarantee that the load balancing thread executes only in the absence of any application threads.

Load distributing algorithms for distributed systems are studied, and their performance is compared in [136]. Many issues concerning load distribution are reviewed here. Some of the key results are: sender-initiated balancers perform well at low to moderate workloads, whereas, the receiver-initiated balancer provides a robust performance at all workloads especially at high workloads, symmetrically (hybrid) balancers are the best choice, and complex load information policies do not necessarily result in good performance. These results are quite intuitive, and are based on a load model which assumes that receiver-initiated work transfers are preemptive. The tasks are independent, and belong to a very general class of applications. The only application dependent feature in choosing a particular balancer is workload. There is no reference to the programming model of the

application, grain size, or architectural parameters like topology, polling interval, etc., in the choice of balancers. Despite the differences in program execution models, the above results are valid for EARTH, and additional studies have been performed regarding the balancers performance under different load conditions, as explained in section 5.

Balter *et al.* [71] argue that contrary to previous reports, the performance benefits of preemptive migration are significantly greater than those of non-preemptive migration, even when the memory transfer cost is high. A distribution of lifetimes of UNIX processes in an academic environment are studied, and this information is used in deriving a preemptive policy. Performance results based on trace-driven simulations are studied to compare this preemptive policy with other preemptive and non-preemptive policies. The migration policy in the load balancing algorithm decides the eligibility of a process for migration as a function of its current age, migration cost, and the loads at its source and target hosts. They suggest that it is preferable to migrate older processes because these processes have a higher probability of living long enough to amortize their migration cost. In the EARTH system this is automatically taken care of by the token queue, which facilitates breadth-first expansion of the activation tree across all the nodes. The reason stated for the better performance of preemptive migration for older processes over non-preemptive migration is the load imbalance caused due to the unpredictability of job execution times with the latter. Further, a preemptive policy is able to make a more accurate prediction about the duration of a process (based on its age)[10] and, more importantly, if the prediction is wrong, it can recover by migrating the process later. While evaluating the migration cost for a process, this model does not evaluate any dependency, locality constraints, and the resultant remote communications these work transfers may spawn in the future. The task model is completely different from that of EARTH, and the results in this paper are not applicable to the EARTH system.

Casavant *et al.* [35] study three distributed dynamic load balancing algorithms - *bi-directional multi-lateral*, *bidding*, and *Bayesian decision or team theory*, with emphasis on the effect of global state information on application performance. They describe simulation experiments which measured the performance and efficiency of distributed algorithms with respect to their reliance on global knowledge. Their results indicate that, for

[10]Current load is the best load predictor. As a rule of thumb, the probability that a process with CPU age of one second uses more than T seconds of total CPU time is 1/T. The age of a process must exceed the migration cost.

the algorithms studied, increasing reliance on dynamically accumulated global information at the expense of reducing response to dynamic system perturbations is rarely beneficial. This work includes the effects of both static and dynamic global knowledge. This results is due to the overhead associated with discrete passing of messages in comparison with the extremely simple objective of load balancing. In addition, their simulation shows that it may be more beneficial to only use information about a small subset of the system which is known to be accurate, than to try to maintain information describing the state of the whole system which may be inaccurate to a greater degree. Besides possible inaccuracy of the collected information, the added time delay in gathering the information hinders performance by slowing response. These results are even more important in the case of very fine-grain multithreaded systems like EARTH, where task grain size may be as small as 12500 cycles. The balance between the overheads in collecting global state information, and the intelligence of balancer decisions based on this information, enables EARTH balancers to scale well for all classes of applications. In contrast to the information policy in this work, EARTH balancers do not exchange periodic load update messages. Instead, remote load information is obtained from piggy-backing normal load balancing messages with load information, history information, and information gleaned from messages that are routed through a node.

The influence of different workload descriptions on a heuristic load balancing scheme is studied in [101]. A task scheduler based on the concept of a stochastic learning automaton [116] on a network OS Unix workstations, is implemented. An artificial, executable workload is created, and a number of experiments are conducted to determine the effect of different workload descriptions. These workload descriptions characterize the load at one host and determine whether a newly created task is to be executed locally or remotely. Typical workload descriptors are: number of tasks in the run queue, size of the free available memory, rate of CPU context switches, rate of system calls, 1-min load average, and the amount of free CPU time. They conclude that, while all the examined workload descriptors lowered the mean response time of tasks when compared to the "no load balancing" case, the best single workload descriptor is the number of tasks in the run queue. Further, combining two of the best descriptors in making load balancing decisions did not result in performance improvements. A similar workload descriptor - number of tokens in the token queue, is employed in the EARTH system.

The Chare [56] system allows users to plug in different load balancing algorithms.

The main load balancer used in their study is ACWN (adaptive contracting-within-neighborhood). It works in a sender-initiated way: when a new chare (similar to tokens in EARTH) is created, the load balancer determines the least loaded neighbor and sends the chare to that node. System load is determined either from load information that is piggy-backed on message packets, or from periodic load status exchanges. Like the other sender-initiated load balancing algorithms, it diffuses the tokens fast when the system load is low, but suffers from unnecessary traffic when the system load is high. For larger grain sizes (10 ms - 1000 ms), *chare* reportedly achieves good performance.

Most of the load balancing studies in distributed computing are based on queuing models. Each node in the network is modeled as a queuing center, with ne w tasks arriving at an average rate λ. However, such models do not accurately match the behavior of multi-threaded architectures, where there is no external arrival of tasks. Rather, a single computation graph is expanded dynamically and some branches are migrated among the nodes to balance the load. The creation of tokens as the graph is expanded cannot be viewed as a random process, as the rate of token generation is linked to the consumption rate.

Compiler and runtime support for adaptive load balancing in software distributed shared memory systems is studied in [89]. Load balancing is studied in workstation environments where the machines might be shared by many users. The compiler is used to provide information that is used to help the run-time system to distribute the work of the parallel loops, not only according to the relative power of the processors, but also in such a way as to minimize communication and page-sharing.

Job scheduling in multiuser environments is of critical importance in large multiprocessor systems. The tasks of a parallel job must be co-scheduled in order to avoid inefficient communication behavior results. Without CO-scheduling, receivers may not be ready when senders are and vice-versa. The gang-scheduling algorithm [120] developed at the Lawrence Livermore Laboratory for the IBM RS/6000 SP system supports time and space sharing of parallel jobs. This policy guarantees that tasks of the same job execute simultaneously.

Coordinated thread scheduling for tightly coupled parallel jobs on workstation clusters running NT, is examined in [32]. This scheduling system coexists with the Windows NT scheduler, and provides coordinated scheduling and can generalize to provide a wide range of resource abstractions. The approach used here is called "demand-based coscheduling".

Job scheduling for multiuser environments in large multiprocessors systems is provided in [54].

The previous work in the EARTH system [33] laid a groundwork for the study on dynamic load balancers in this thesis. This thesis can be seen as an extension or logical follow-up of [33, 34]. The previous results show that a hybrid, history information based balancer provides the best possible performance for most classes of applications. Further, the results show that it is impossible to build a single load balancer for all classes of applications. A set of program, architectural and balancer related parameters like grain size, application model, polling interval, logical topology, balancer algorithm determine program performance. In this thesis, we add few more features to this list like workload, number of nodes, quality of load state information, message complexity, non-intrusiveness of the balancer from application execution, ratio of network speed to CPU speed, network bandwidth, network interface in the runtime system. Further, we show that load state information performs better than history information, and it is possible to tolerate the overheads from load information gathering actions, and randomizing functions, and still achieve scalable, robust performance for fine-grain applications. We base the *Rand* balancer on existing theoretical proof for distributed computing models, and show that it does outperform the best balancers suggested in [33]. We implement a new balancer, *Minima* that provides a realistic lower bound for parallel performance and compare it to the *Nop* balancer proposed earlier. Finally, we clearly identify appropriate balancers for different applications and load situations. While reviewing the results in [33], it must be noted that the CPU speed in that case was 62.5 MHz, whereas our results are based on a 120 MHz CPU. This increase in speed decreases the sequential execution time considerably, making it more challenging to achieve linear speedups.

We presented the distinction between fine grain multi-threading systems that evolved from the classical data-flow model, and operating system based multi-threading systems that are a refinement of the concept of a process in operating system. The first exploits parallelism at a finer grain and has a lower thread switching cost than the later. Both systems find appropriate areas of applications. We then described several implementations of multi-threading systems implemented in each category, and referred to the dynamic load balancing work in these systems. Finally we reviewed the work done in the area of dynamic load balancing in distributed computing so far.

Bibliography

[1] CAPSL tech. memo, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[2] Tech. rep. mit/lcs/tr-, MIT Lab. for Comp. Sci.

[3] *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.

[4] *Proc. of the IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, PACT '95*, Limassol, Cyprus, Jun. 1995. ACM Press.

[5] *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT '96)*, Boston, Mass., Oct. 1996. IEEE Comp. Soc. Press.

[6] *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation*, Montréal, Qué., Jun. 1998.

[7] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large scale distributed-memory multiprocessor. Tech. Memo MIT/LCS/TM-454, MIT Lab. for Comp. Sci., 1991.

[8] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 System Architecture. In *IBM Systems Journal, Reprint Order No. G321 - 5563*, volume 34, 1995.

[9] Rakesh Agrawal and Ahmed K. Ezzat. Location Independent Remote Execution in NEST. In *IEEE Transactions on Software Engineering*, volume 13, pages 905–913, August 1987.

[10] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proc. of the 31st Ann. Intl. Symp. on Microarchitecture*, pages 226–236, Dallas, Tex., Nov.–Dec. 1998.

[11] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proc., 1990 Intl. Conf. on Supercomputing, Amsterdam, The Netherlands*, pages 1–6, Jun. 1990.

[12] Makoto Amamiya. An ultra-multiprocessing architecture for functional languages. In Gaudiot and Bic [68], chapter 3, pages 95–119. Book contains papers presented at the First Workshop on Data-Flow Computing, Eilat, Israel, May 1989.

[13] Makoto Amamiya, Tetsuo Kawano, Hiroshi Tomiyasu, and Shigeru Kusakabe. A practical processor design for multithreading. In *Proc. of Frontiers '96: The Sixth Symp. on the Frontiers of Massively Parallel Computation*, pages 23–32, Annapolis, Mary., Oct. 1996.

[14] José Nelson Amaral, Guang R. Gao, Phillip Merkey, Thomas Sterling, Zachary Ruiz, and Sean Ryan. An HTMT performance prediction case study: Implementing Cannon's dense matrix multiply algorithm. CAPSL Tech. Memo 26, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Feb. 1999. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[15] Boon Seong Ang, Arvind, and Derek Chiou. StarT the Next Generation: Integrating global caches and dataflow architecture. CSG Memo 354, Computation Structures Group, MIT Lab. for Comp. Sci., Aug. 1994.

[16] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proc. of the Tenth Annual ACP Symposium on Parallel Algorithms and Architectures, Puerto vallarta, Mexico*, pages 119–129, June-July 1998.

[17] Arvind and Kim P. Gostelow. The U-Interpreter. *Computer*, 15(2):42–49, Feb. 1982.

[18] W. C. Athas and C. L. Seitz. Multicomputers: Message-Passing Concurrent Computers. In *Computer*, volume 21, pages 9–24, August 1988.

[19] Eduard Ayguade', Mario Furnari, Maurizio Giordano, Hans-Christian Hoppe, Jesus Labarta, Xavier Martorell, Nacho Navarro, Dimitrios Nikolopoulos, Theodore Papatheodorou, and Eleftherios Polychronopoulos. Nano-Threads: Programming Model Specification. In *Deliverable M1.D1, ESPRIT Project NANOS (No. 21907), University of Patras*, Jul. 1997.

[20] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced Allocations. In *Proc. of the 26th Ann. Symp. Theory of Computing*, pages 593–602, May 1994.

[21] Hannah Bast. Dynamic Scheduling with Incomplete Information. In *Proc. of the Tenth Annual ACP Symposium on Parallel Algorithms and Architectures, Puerto vallarta, Mexico*, pages 182–191, June-July 1998.

[22] Petra Berenbrink, Tom Friedetzky, and Ernst W. Mayr. Parallel Continous Randomized Load Balancing. In *Proc. of the Tenth Annual ACP Symposium on Parallel Algorithms and Architectures. Puerto vallarta, Mexico*, pages 192–201, June-July 1998.

[23] Daniel J. Berg. Java Threads. In *A White Paper, Sun Microsystems, California, USA*, pages 109–114, March 1996.

[24] Robert Blumofe and Charles Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proc. of the 35th Annual Symposium on foundations of Computer Science (FOCS), Santa Fe, New Mexico*, pages 356–368, Nov. 1994.

[25] Robert D. Blumofe. Executing Multithreaded Programs Efficiently. Technical report, Laboratory of Computer Science, Massachussetts Institute of Technology, Boston, USA, 1995. PhD thesis, 1995.

[26] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. of the 8th Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, Jun. 1996.

[27] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Journal of Parallel and Distributed Computing*, volume 37, pages 55–69, Aug. 1996.

[28] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the Fifth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pages 207–216, Santa Barbara, Calif., Jul. 1995.

[29] Nanette Jackson Boden. Runtime Systems for Fine-Grain Multicomputers. In *Ph.D Thesis, Department of Computer Science, California Institute of Technology, Pasadena, California (also available as Technical Report - Caltech-CS-TR-92-10)*, 1993.

[30] Bob Boothe and Abhiram Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture* [3], pages 214–223.

[31] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 704–709, Cancún, Mexico, Apr. 1994. IEEE Comp. Soc.

[32] Matt Buchanan and Andrew A. Chien. Coordinated Thread Scheduling for Workstation Clusters Under Windows NT. In *Technical Report, Concurrent Systems Architecture Group, Department of Computer Science, University of Illinois, Urbana-Champaign*, 1999.

[33] Haiying Cai. Dynamic load balancing on the EARTH-SP system. Master's thesis, McGill U., Montréal, Qué., May 1997.

[34] Haiying Cai, Olivier Maquelin, Prasad Kakulavarapu, and Guang R. Gao. Design and Evaluation of Dynamic Load Balancing Schemes under a Fine-grain Multithreaded Execution model. In *Proc. of the Multithreaded Execution Architecture and Compilation Workshop, Orlando, Florida*, Jan. 1999.

[35] Thomas L. Casavant and Jon G. Kuhl. Analysis of Three Dynamic Distributed Load-Balancing Strategies with Varying Global Information Requirements. In *IEEE Computer*, pages 185–192, August 1987.

[36] Gerson G. H. Cavalheiro, Francois Galilee, and Jean-Louis Roch. Athapascan-1: Parallel programming with asynchronous tasks. In *Technical Report, LMC-IMAG-APACHE Project, Grenoble, Franca, http://www-apache.imag.fr*, 1999.

[37] Cornell Theory Center. The SP2 Switch. In *Documentation at http://www.tc.cornell.edu/Edu/Talks/SP/switch.html.*

[38] Cornell Theory Center. What are LoadLeveler, EASY, and EASY-LL. In *Documentation at http://www.tc.cornell.edu/UserDoc/ SP/Batch/what.html.*

[39] Soumen Chakrabarti. Efficient Resource Scheduling in Multiprocessors. In *Ph.D, University of California, Berkeley*, 1996.

[40] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting Data Races in Cilk Programs that Use Locks. In *Proc. of 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'98), Puerto Val larta, Mexico*, pages 298–309, June 1998.

[41] Nikos Chrisochoides. Multithreaded Model for Dynamic Load Balancing Parallel Adaptive PDE Computations. In *Technical Report, Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, New York*, 1994.

[42] S. Crane. The REX Lightweight Process Library. In *Computer Science Technica Report, Imperial College of Science and Technology, London, England*, 1993.

[43] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of the Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA*, Apr. 1991.

[44] C.Xu and F.Lau. Load Balancing in Parallel Computers. In *Kluwer Academic Publishers, Boston,MA*, 1997.

[45] Jack B. Dennis. First version of a data-flow procedure language. In *Proc. of the Colloque sur la Programmation*, number 19 in Lec. Notes in Comp. Sci., pages 362–376, Paris, France, Apr. 9–11, 1974. Springer-Verlag.

[46] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: Principles, projects, and issues. In Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, editors, *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 1, pages 1–72. Kluwer Academic Pub., Norwell,

Mass., 1994. Book contains papers presented at the Workshop on Multithreaded Computers, Albuquerque, N. Mex., Nov. 1991.

[47] Jack B. Dennis, Guang-Rong Gao, and Kenneth W. Todd. Modeling the weather with a data flow supercomputer. *IEEE Trans. on Computers*, 33(7):592–603, Jul. 1984.

[48] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. In *Software - Practice and Experience*, volume 21, pages 757–785, August 1991.

[49] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *Proc. of the IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, PACT '95* [4], pages 109–121.

[50] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. In *IEEE Transaction on Software Engineering*, volume 12, pages 662–675, May 1986.

[51] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. In *Performance Evaluation*, volume 6, pages 53–68, 1986.

[52] Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebe cca Stamm, and Dean Tullsen. Simultaneous Multithreading: A Platform for Next-generation Processors. In *Proc. of IEEE Micro*, pages 12–18, sept 1997.

[53] D. G. Feitelson and L. Rudolph. Toward Convergnce in Job Schedulers for Parallel Supercomputers. In *In* Job Scheduling Strategies for Parallel Processing, *D. G. Feitelson and L. Rudolph (eds.), Springer Verlag, Lecture Notes in Computer Science*, volume 1162, pages 1–26, 1996.

[54] Dror G. Feitelson. Job Scheduling in Multiprogrammed Parallel Systems. In *Technical Report, Institute of Computer Science, The Hebrew University, 91904 Jerusalem, Israel. Original version of this work done at IBM T. J. Watson Research Center, Yorktown Heights, NY 10598*, August 1997.

[55] Edward W. Felten and Dylan McNamee. Improving the performance of message-passing applications by multithreading. In *Proc. of the Scalable High Performance Computing Conf. (SHPCC-92)*, pages 84–89, Williamsburg, Virginia, Apr. 26–29, 1992. IEEE Comp. Soc.

[56] W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha, and L.V. Kale. Supporting Machine Independent Programming on Diverse Parallel Architectures. In *Proc. of the International Conference on Parallel Processing*, pages 193–201, Aug. 1991.

[57] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. of Parallel and Distrib. Computing*, 10(4):349–366, Dec. 1990.

[58] Adam Ferrari and V. S. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *Technical Report, Dept. of Mathematics and Computer Science, Emory University, Atlanta, GA, USA*, 1994.

[59] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-grain Parallelism on a Cluster of Workstations. In *Proc. of the First Symposium on Operating Systems Design and Implementation, Usenix Association*, Nov. 1994.

[60] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1998.

[61] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation* [6], pages 212–223.

[62] Guang Gao, José Nelson Amaral, Andres Marquez, and Kevin Theobald. A refinement of the HTMT program execution model. CAPSL Tech. Memo 22, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Jul. 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[63] Guang R. Gao. An Efficient Hybrid Dataflow Architecture Model. In *Journal of Parallelism*, volume 19, Dec. 1993.

[64] Guang R. Gao, José Nelson Amaral, Andrés Márquez, Kevin B. Theobald, Sean Ryan, Zachary Ruiz, Thomas Geiger, and Christopher J. Morrone. HTMT phase 2

report. CAPSL Tech. Memo 31, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Jul. 1999. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[65] Guang R. Gao, Lubomir Bic, and Jean-Luc Gaudiot, editors. *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Comp. Soc. Press, 1995. Book contains papers presented at the Second Intl. Work. on Dataflow Computers, Hamilton Island, Australia, May 1992.

[66] Guang R. Gao, Herbert H. J. Hum, and Yue-Bong Wong. Parallel Function Invocation in a Dynamic Argument-Fetching Dataflow Architecture. In *Proc. of PARBASE-90: Intl. Conf. on Databases, Parallel Architectures, and their Applications, Miami Beach, Florida*, pages 112–116, Mar. 1990.

[67] Guang R. Gao, Kevin B. Theobald, Andrés Márquez, and Thomas Sterling. The HTMT program execution model. CAPSL Tech. Memo 09, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Jul. 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[68] Jean-Luc Gaudiot and Lubomir Bic, editors. *Advanced Topics in Data-Flow Computing*. Prentice-Hall, Englewood Cliffs, N. Jer., 1991. Book contains papers presented at the First Workshop on Data-Flow Computing, Eilat, Israel, May 1989.

[69] James Gosling and Henry McGilton. The Java Language Environment. In *A White Paper, Sun Microsystems, California, USA*, pages 1–95, May 1996.

[70] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Comm. of the ACM*, 28(1):34–52, Jan. 1985.

[71] Mor Harchol-Balter and Allen B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. In *Proceeedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Philadelphia, PA*, pages 13–24, May 1996.

[72] Gerd Heber, Rupak Biswas, Parimala Thulasiraman, and Guang R. Gao. Using multithreading for the automatic load balancing of adaptive finite element meshes. In *Proc. of the 5th Intl. Symp. on Solving Irregularly Structured Problems in Parallel*, number 1457 in Lec. Notes in Comp. Sci., pages 132–143, Berkeley, Calif., Aug. 1998. Springer-Verlag.

[73] Alan Heirich and Stephen Taylor. A Parabolic Load Balancing Method. In *Technical Report, Scalable Concurrent Programming Laboratory, California Institute of Technology*, 1993.

[74] L. J. Hendren, X. Tang, Y. Zhu, G. R. Gao, X. Xue, H. Cai, and P. Ouellet. Compiling C for the Earth Multithreaded Architecture. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT'96), Boston, Mass.Intl. Journal of Parallel Programming*, pages 12–23, Oct. 1996.

[75] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT '96)* [5], pages 12–23.

[76] Sébastien Hily and André Seznec. Branch prediction and simultaneous multithreading. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT '96)* [5], pages 169–173.

[77] Kei Hiraki, Satoshi Sekiguchi, and Toshio Shimada. Status report of SIGMA-1: A data-flow supercomputer. In Gaudiot and Bic [68], chapter 7, pages 207–223. Book contains papers presented at the First Workshop on Data-Flow Computing, Eilat, Israel, May 1989.

[78] R. Hofman and W. G. Vree. Distributed Hierarchical Scheduling with Explicit Grain Size Control. In *Future Generation Computer Systems*, volume 8, pages 111–119, July 1992.

[79] Allen Holub. Progrmmaing Java threads in the real world. In *Java Threads Series, JavaWorld, http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads.html*, pages 109–114, Feb. 1996.

[80] Cay S. Horstmann and Gary Cornell. Core JAVA. In *Vol. I and II, The Java Series, Sun Microsystems, California, USA*, 1997.

[81] G. Horton. A multi-level diffusion method for dynamic load balancing. In *Parallel Computing Journal*, volume 19, pages 209–218, 1993.

[82] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *Intl. J. of Parallel Programming*, 24(4):319–347, Aug. 1996.

[83] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. The Multi-Threaded Architecture multiprocessor. ACAPS Tech. Memo 88, Sch. of Comp. Sci., McGill U., Montréal, Qué., Dec. 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[84] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In *Proc. of the IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, PACT '95* [4], pages 59–68.

[85] Herbert Hing-Jing Hum. *The Super-Actor Machine: a Hybrid Dataflow/von Neumann Architecture*. PhD thesis, McGill U., Montréal, Qué., May 1992.

[86] Robert A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proc. of the 15th Ann. Intl. Symp. on Computer Architecture*, pages 131–140, Honolulu, Haw., May–Jun. 1988.

[87] IBM. Interconnection Technologies for High-Performance Computing (RS/6000 SP). In *Documentation at http://www.rs6000.ibm.com/resource/technology/sp-sw2/spswp2-1.html.*

[88] IBM. The RS/6000 SP High-Performance Communication Network. In *Documentation at http://www.rs6000.ibm.com/resource/technology/sp-sw1/spswp1.book-1.html.*

[89] Sotiris Ioannidis and Sandhya Dwarakadas. Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems. In *Technical Report, Department of Computer Science, Univeristy of Rochester, Rochester, New York 14627-0226*, 1997.

[90] JavaSoft. The JAVA HOTSPOT Performance Engine Architecture. In *A White Paper, Sun Microsystems, California, USA, http://www.java.sun.com/products/hotspot/whitepaper.html*, April 1999.

[91] Prasad Kakulavarapu and José Nelson Amaral. A survey of load balancers in modern multi-threading systems. In *Proc. of the 11th Symp. on Computer Architecture and High Performance Computing*, pages 10–16, Natal, Brazil, Sep.–Oct. 1999.

[92] Prasad Kakulavarapu, Olivier Maquelin, and Guang R. Gao. Design of the runtime system for the Portable Threaded-C language. CAPSL Tech. Memo 24, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Jul. 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[93] Prasad Kakulavarapu, Christopher J. Morrone, Kevin B. Theobald, José Nelson Amaral, and Guang R. Gao. A Comparitive Study of Multithreaded Environment on Distributed Memory Machines. In *To appear in Proc. of the 19th IEEE Intl. Performance, Computing, and Communications Conference-IPCCC 2000, Embassy Suites Phoenix North, Phoenix, Arizona, USA*, February 2000.

[94] Prasad Kakulavarapu, Christopher J. Morrone, Kevin B. Theobald, José Nelson Amaral, and Guang R. Gao. A Comparitive Study of Multithreaded Environment on Distributed Memory Machines. In *CAPSL Technical Memo 35, University of Delaware, Newark, Delaware, USA*, November 1999.

[95] Vijay Karamcheti. Run-time techniques for dynamic multithreaded computations. In *Ph.D Thesis, Department of Electrical Engineering, University of Illinois at Urbana-Champaign*, 1998.

[96] Vijay Karamcheti, John Plevyak, and Andrew A. Chien. Runtime Mechanisms for Efficient Dynamic Multithreading. In *Journal of Parallel and Distributed Computing*, volume 37, pages 21–40, Aug. 1996.

[97] Tetsuo Kawano, Shigeru Kusakabe, Rin ichiro Taniguchi, and Makoto Amamiya. Fine-grain multi-thread processor architecture for massively parallel processing. In *Proc. of the First Intl. Symp. on High-Performance Computer Architecture*, pages 308–317, Raleigh, N. Caro., Jan. 1995.

[98] Ashfaq A. Khokhar, Gerd Heber, Parimala Thulasiraman, and Guang R. Gao. Load adaptive algorithms and implementations for the 2D discrete wavelet transform on fine-grain multithreaded architectures. In *Proc. of the 13th Intl. Parallel Processing Symp. and the 10th Symp. on Parallel and Distributed Processing*, pages 458–462, San Juan, Puerto Rico, Apr. 1999. IEEE Comp. Soc. and ACM SIGARCH.

[99] D. Klappholz and H-C. Park. Parallelized Process Scheduling for a Tightly-Coupled MIMD Machine. In *Proc. of the Intl. Conf. on Parallel Processing*, pages 315–321, August 1984.

[100] Douglas Kramer. The Java Platform. In *A White Paper, Sun Microsystems, California, USA*, pages 1–24, May 1996.

[101] Thomas Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. In *IEEE Transactions on Software Engineering*, volume 17, pages 725–730, July 1991.

[102] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proc. of the Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, San Jose, Calif., Oct. 1994.

[103] Doug Lea. Concurrent Programming in Java - Design Principles and Patterns. In *Addison-Wesley*, 1997.

[104] Bil Lewis and Daniel J. Berg. Threads Primer - A Guide to Multithreaded Programming. In *Sunsoft Press, Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100, USA*, 1996.

[105] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. In *The Java Series, Sun Microsystems, California, USA*, 1997.

[106] Mat Loikkanen and Nader Bagherzadeh. A fine-grain multithreading superscalar architecture. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT '96)* [5], pages 163–168.

[107] Olivier Maquelin. The ADAM architecture and its simulation. TIK-Schriftenreihe 4, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, Switzerland, 1994. PhD thesis, 1994.

[108] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proc. of the 23rd Ann. Intl. Symp. on Computer Architecture*, pages 178–188, Philadelphia, Penn., May 1996.

[109] Olivier C. Maquelin. Load balancing and resource management in the ADAM machine. In Gao et al. [65], pages 307–323. Book contains papers presented at the Second Intl. Work. on Dataflow Computers, Hamilton Island, Australia, May 1992.

[110] Olivier C. Maquelin. Load balancing and resource management in the adam machine. In *Second Workshop on Dataflow Computing, Hamilton Island, Australia, 1992, Published in Advanced Topics in Dataflow Computing and Multithreading, Lubomir Bic, Guang R. Gao, Jean-Luc Gaudiot editors, IEEE Computer Society*, 1995.

[111] Olivier C. Maquelin, Herbert H. J. Hum, and Guang R. Gao. Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors. In *Proc. of the First Intl. EURO-PAR Conference, no. 966 in Lecture Notes in Computer Science, Stockholm, Sweden*, pages 117–128, Aug. 1995.

[112] Olivier C. Maquelin, Herbert H. J. Hum, and Guang R. Gao. Costs and benefits of multithreading with off-the-shelf RISC processors. In *Proc. of the First Intl. EURO-PAR Conf.*, number 966 in Lec. Notes in Comp. Sci., pages 117–128, Stockholm, Sweden, Aug. 1995. Springer-Verlag.

[113] Edward Mascarenhas and Vernon Rego. Ariadne: Architecture of a Portable Threads system supporting Thread Migration. In *Software - Practice and Experience*, volume 26(3), pages 327–356, Mar. 1996.

[114] Piyush Mehrotra and Matthew Haines. An Overview of the Opus Language and Runtime System. Technical report, May 1994.

[115] Paul Messina, David Culler, Wayne Pfeiffer, William Martin, J. Tinsley Oden, and Gary Smith. Architecture - The High-Performance Computing Continuum. In *Communications of the ACM*, volume 41, pages 36–44, November 1998.

[116] Ravi Mirchandaney and John A. Stankovic. Using a Stochastic Learninmg Automaton for Job Scheduling in Distributed Processing Systems. In *Journal of Parallel and Distributed Computing*, pages 527–551, 1986.

[117] Ravi Mirchandaney, Don Towsley, and John A. Stankovic. Adaptive Load Sharing in Heterogeneous Distributed Systems. In *Journal of Parallel and Distributed Computing*, number 9, pages 331–346, 1990.

[118] Michael Mitzenmacher. Analyses of Load Stealing Models Based on Differential Equations. In *Proc. of the Tenth Annual ACP Symposium on Parallel Algorithms and Architectures, Puerto vallarta, Mexico*, pages 212–221, June-July 1998.

[119] Michael David Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. In *Ph. D Thesis, University of California, Berkeley, California*, 1996.

[120] J. E. Moreira, H. Franke, W. Chan, and L. L. Fong. A Gang-Scheduling System for ASCI Blue-Pacific. In *Technical Report RC 21359 (96204), IBM Research Division*, December 1998.

[121] Frank Mueller. Distributed Shared-Memory Threads: DSM-Threads. In *Technical Report, Work in Progress, Humboldt-Universitat zu Berlin, Institut fur Informatik, 10099 Berlin, Germany*, 1998.

[122] S. Muthukrishnan and Rajmohan Rajaraman. An Adversial Model for Distributed Dynamic Load Balancing. In *Proc. of the Tenth Annual ACP Symposium on Parallel Algorithms and Architectures, Puerto vallarta, Mexico*, pages 47–54, June-July 1998.

[123] W. Najjar and J.-L. Gaudiot. Multi-level execution in data-flow architectures. In *Proc. of the 1987 Intl. Conf. on Parallel Processing*, pages 32–39, St. Charles, Ill., Aug. 1987.

[124] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture* [3], pages 156–167.

[125] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. of the 16th Ann. Intl. Symp. on Computer Architecture*, pages 262–272, Jerusalem, Israel, May–Jun. 1989.

[126] Scott Oaks and Henry Wong. Java Threads. In *First Edition, O'Reilly and Associates, USA*, Jan. 1997.

[127] Gregory Michael Papadopoulos. Implementation of a general purpose dataflow multiprocessor. Tech. Rep. MIT/LCS/TR-432, MIT Lab. for Comp. Sci., Aug. 1988. PhD thesis.

[128] Hisham Petry. Earth Threaded-C Programming Manual. Technical report, Mar. 1996.

[129] C. Gary Rommel. The Probability of Load Balancing Success in a Homogeneous Network. In *IEEE Transactions on Software Engineering*, volume 17, pages 922–933, September 1991.

[130] James Rumbaugh. A data flow multiprocessor. *IEEE Trans. on Computers*, 26(2):138–146, Feb. 1977.

[131] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Technical Report, Computer Science Division, University of California, Berkeley, California 94720*, 1995.

[132] Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Conf. Proc., 1993 Intl. Conf. on Supercomputing*, pages 251–260, Tokyo, Japan, Jul. 1993.

[133] Mitsuhisa Sato, Yuetsu Kodama, Suichi Sakai, Yoshinori Yamaguchi, and Yasuhito Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture* [3], pages 146–155.

[134] Klaus Eric Schauser, David E. Culler, and Thorsten von Eiken. Compiler-controlled multithreading for lenient parallel languages. Rep. No. UCB/CSD 91/640, Comp. Sci. Div., U. of Calif. at Berkeley, 1991.

[135] Klaus Erik Schauser, David E. Culler, and Thorsten von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proc. of FPCA '91 Conference on Functional Programming Languages and Computer Architecture, Spring er Verlag*, aug 1991.

[136] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. In *IEEE Computer*, pages 33 – 44, December 1992.

[137] Andrew Sohn, Chinhyun Kim, and Mitsuhisa Sato. Multithreading with the EM-4 distributed-memory multiprocessor. In *Proc. of the IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, PACT '95* [4], pages 27–36.

[138] Andrew Sohn, Mitsuhisa Sato, Namhoon Yoo, and Jean-Luc Gaudiot. Effects of multithreading on data and workload distribution for distributed-memory multiprocessors. In *Proc. of the 10th Intl. Parallel Processing Symp.*, Honolulu, Haw., Apr. 1996. IEEE Comp. Soc. and ACM SIGARCH.

[139] Bin Song. Scheduling Adaptively Parallel Jobs. In *Masters Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Boston*, January 1998.

[140] Vason P. Srini. An architectural comparison of dataflow systems. *Computer*, 19(3):68–88, Mar. 1986.

[141] T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, 1999.

[142] Harold S. Stone. High-Performance Computer Architecture. In *Addison-Wesley Publishing Company, Massachusetts*, 1993.

[143] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Pub. Co., 3rd edition, 1993.

[144] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. In *Journal of Concurrency: Practice and Experience*, volume 2(4), pages 315–339, December 1990.

[145] V. S. Sunderam. TPVM: A Threads-Based Interface and Subsystem for PVM. In *Draft Version, Dept. of Mathematics and Computer Science, Emory University, Atlanta, GA, USA*, June 1994.

[146] Xinan Tang, Olivier Maquelin, Kevin B. Theobald, Guang R. Gao, and Prasad Kakulavarapu. A portable Threaded-C language for EARTH multiprocessors.

CAPSL Tech. Note 02, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Jan. 1998.

[147] Kenjiro Taura. Efficient and Reusable Implementation of Fine-Grain Multithreading and Garbage Collection on Distributed-Memory Parallel Computers. In *Ph.D Thesis, Department of Information Science, University of Tokyo*, 1997.

[148] Kenjiro Taura and Akinori Yonezawa. Fine-grain multithreading with minimal compiler support — a cost effective approach to implementing efficient multithreading languages. In *Proc. of the ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, pages 320–333, Las Vegas, Nev., Jun. 1997.

[149] Scott R. Taylor. A comparison of multithreading implementations. In *Proc. of the Yale Multithreaded Programming Work.*, New Haven, Conn., Jun. 8–9, 1998.

[150] Kevin B. Theobald. EARTH - an Efficient Architecture for Running THreads. Technical report, School of Computer Science, McGill University, Montreal, Québec, 1999. PhD thesis, 1999.

[151] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Tech. Memo 19, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Mar. 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[152] Kevin B. Theobald, Jose Nelson Amaral, Gerd Herber, Oliver Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C Language. In *Technical Memo 19, CAPSL Lab, University of Delaware*, Mar. 1998.

[153] Kevin B. Theobald and Guang R. Gao. The Benefits of Hardware-Assisted Fine-Grain Multithreading. In *Technical Memo 32, CAPSL Lab, University of Delaware, Newark, Delaware, USA*, pages 1–27, July 1999.

[154] Kevin B. Theobald, Guang R. Gao, and Thomas L. Sterling. Superconducting processors for HTMT: Issues and challenges. In *Proc. of Frontiers '99: The 7th Symp. on the Frontiers of Massively Parallel Computation*, pages 260–267, Annapolis, Mary., Feb. 1999.

[155] Kevin Bryan Theobald. Adding fault-tolerance to a static data flow supercomputer. Tech. Rep. MIT/LCS/TR-499, MIT Lab. for Comp. Sci., Apr. 1991. Master's thesis, Dec., 1990.

[156] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill U., Montréal, Qué., May 1999.

[157] John Thornley, K. Mani Chandy, and Hiroshi Ishii. A System for Structured High-Performance Multithreaded Programming in Windows NT. In *Proc. of the 2nd USENIX Windows NT Symposium, pp. 67-76, Seattle, Washington*, Aug. 1998.

[158] Ruppa K. Thulasiram and Guang R. Gao. Option Pricing Problem on a Multithreaded Parallel Architecture. In *CAPSL Technical Memo 25, University of Delaware, Newark, Delaware, USA*, November 1998.

[159] Ruppa K. Thulasiram and Guang R. Gao. A Multithreaded Parallel Computational Approach for Valuing Derivatives. In *Proc. of the 1st WAFA Conference, George Mason University*, April 1999.

[160] Xinmin Tian, Olivier Maquelin, Xinan Tang, Kevin Theobald, Guang R. Gao, and Herbert H.J. Hum. The Mcgill Earth Benchmark Suite EBS. Technical report, 1996.

[161] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proc. of the Fifth Intl. Symp. on High-Performance Computer Architecture*, pages 54–58, Orlando, Flor., Jan. 1999.

[162] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism. In *Proc. of the Tenth Annual ACP Symposium on Parallel Algorithms and Architectures, Puerto vallarta, Mexico*, pages 140–151, June-July 1998.

[163] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Intl. Symposium on Computer Architecture, ACM Press, Gold Coast, Australia*, May 1992.

[164] Jerrell Watts and Stephen Taylor. A Practical Approach to Dynamic Load Balancimg. In *Technical Report, Scalable Concurrent Programming Laboratory, Syracuse University, Syracuse, New York*, December 1997.

[165] Jerrell Watts and Stephen Taylor. A practical approach to dynamic load balancing. In *IEEE Transaction on Parallel and Distributed Systems*, volume 9, Mar. 1998.

[166] Jerrell Watts and Stephen Taylor. A Practical Approach to Dynamic Load Balancing. In *IEEE Transaction on Parallel and Distributed Systems*, volume 9, Mar. 1998.

[167] Boris Weissman. Active Threads: an Extensible and Portable Light-Weight Thread System. Technical report, Sep. 1997.

[168] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. In *IEEE Transaction on Parallel and Distributed Systems*, volume 4, No. 9, pages 979–993, Sep. 1993.

[169] Mohammed Javeed Zaki, Wei Li, and Srinivasan Parthasarathy. Customized Dynamic Load Balancing for a Network of Workstations. In *Technical Report 602, Computer Science Department, University of Rochester, Rochester, New York 14627*, pages 1–23, December 1995.

[170] Yingchun Zhu and Laurie J. Hendren. Communication optimizations for parallel C programs. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation* [6], pages 199–211.

Appendix A

EARTH Primitives in Threaded-C

This appendix gives a complete list of all the EARTH Threaded-C primitives and briefly explains how they are used.

A.1 Threads and Functions

THREADED

Keyword for a threaded function declaration.

THREAD_*n*

Marks the beginning of a thread, *n* is an integer value greater than 0. This number labels a thread within a function.

void END_THREAD(void)

Marks the end of a thread. Control then switches to another ready thread.

int NUM_NODES

Run-time system variable set to the number of available nodes in the system.

int NODE_ID

Run-time system variable set to the local node number. This number ranges from 0 to NUM_NODES - 1.

void POLL(void)

Polls the network and handles any available messages. Together with the NUM_NODES and NODE_ID primitives, this is one of the only primitives that

can be used from non-threaded functions. Inserting POLL statements into long threads can significantly improve overall performance, as external requests are handled more quickly.

void CALL(func_name, ...)

Calls function *func_name* sequentially and blocks until that function terminates. Functions invoked with CALL must terminate with RETURN instead of END_FUNCTION.

void RETURN(void)

Ends a function that will be called with the CALL primitive. This tells the compiler to generate sequential entry/exit code.

void INVOKE(int node, func_name, ...)

The programmer specifies the processing node on which the function *func_name* will be executed. Function *func_name* must terminate with END_FUNCTION.

void TOKEN(func_name, ...)

Similar to INVOKE, but it is the runtime system that decides on which node the function will execute. Function *func_name* must terminate with END_FUNCTION.

void END_FUNCTION(void)

Marks the end of a threaded function that is called with INVOKE or TOKEN.

A.2 Thread Synchronization

Threads are often associated with a synchronization slot. The sync count in that slot represents how many signals the thread has to wait for before it can be activated. The programmer can initialize the sync count and update the count to control the firing of a thread. We use the following EARTH primitives to operate on the sync slots:

SLOT

A pre-defined type for synchronization slots.

SLOT SYNC_SLOTS[N]

This is how synchronization slots have to be declared at the beginning of a function.

SPTR

A pre-defined type defined as: **typedef SLOT *GLOBAL SPTR.**

void *GLOBAL FRAME_ADR(void)

Returns a global pointer to the current frame.

void *IP_ADR(int thread_num)

Returns a (local) pointer to the first instruction of thread *thread_num*. This does not have to be a global pointer, as each node has a copy of the program code located at the same addresses.

SPTR SLOT_ADR(int slot_num)

Returns a global pointer to sync slot *slot_num*.

void INIT_SYNC(int slot_num, int init_count, int reset_count, int thread_num)

Initializes sync slot *slot_num* with the initial counter value *init_count*, the reset value *reset_count*, and a thread pointer for thread *thread_num*.

void SYNC(int slot_num)

Decreases the sync count of slot *slot_num* by one. If the count reaches zero the corresponding thread is scheduled for execution.

void RSYNC(SPTR slot_adr)

Same as SYNC(), but the sync slot is specified by a global address.

void INCR_SYNC(int slot_num, int val)

Increases the sync count of slot *slot_num* by *val*. If the count becomes zero the corresponding thread is scheduled for execution.

void INCR_RSYNC(SPTR slot_adr, int val)

Same as INCR_SYNC(), but the sync slot is specified by a global address.

void SPAWN(int thread_num)

Schedules local thread *thread_num* for execution.

void RSPAWN(void *GLOBAL FP, void *IP)

Same as SPAWN(), but the thread is specified explicitly with its frame and instruction pointers.

Implicit sync operation

All data transfer primitives also perform a sync operation after the data has reached its destination.

A.3 Data Transfer Primitives

The data transfer primitives support remote memory accesses and block data tranfers. Short data transfers of single bytes or words of memory are supported by the GET_SYNC_x and DATA_SYNC_x. Several versions of these primitives exist, which are distinguished by their suffix. For example, the suffix _L is used for 32-bit (long word) values. Here is the complete list of suffixes:

_B (char): Single byte (8 bits).

_S (short): Short word (16 bits).

_L (long): Long word (32 bits).

_F (float): Float size (32 bits).

_D (double): Double size (64 bits).

_G (void *GLOBAL): Global pointer (either 32 or 64 bits).

In addition, the sync slot that should be signalled when the opeation terminates can either be specified as a (local) slot number (_SYNC_ variants), or as a global pointer (_RSYNC_ variants). Here are the basic communication primitives:

void DATA_SYNC_x(T datum, void *dest, int slot_num)

Sends a value to the destination address and then update the specified sync slot. The type of the value has to be assignment compatible to either a byte, a short, a long, a float, a double or a global pointer.

void DATA_RSYNC_x(T datum, void *dest, SPTR slot_adr)

Same as DATA_SYNC_x(), but the sync slot is specified as a global address.

void GET_SYNC_x(void *GLOBAL src, void *GLOBAL dest, int slot_num)

Reads a value from the source address and copy it to the destination address. Then, update the specified sync slot.

void GET_RSYNC_x(void *GLOBAL src, void *GLOBAL dest, SPTR slot_adr)

Same as GET_SYNC_x(), but the sync slot is specified as a global address.

void BLKMOV_SYNC(void *GLOBAL src, void *GLOBAL dest, long length, int slot_num)

Copies *length* bytes of data from the source to the destination address and updates the specified sync slot.

void BLKMOV_RSYNC(void *GLOBAL src, void *GLOBAL dest, long length, SPTR slot_adr)

Same as BLKMOV_SYNC(), but the sync slot is specified as a global address.

A.4 Global Address Support

GLOBAL

Type qualifier used to distinguish global pointers (64-bit entities) from local (normal) pointers.

T *GLOBAL TO_GLOBAL(T *ptr)

Turns a local pointer into a global pointer that points to address ptr on the local node. In the portable implementation the type of the result depends on the type of the argument. On MANNA, the result is of type pointer to void.

T *TO_LOCAL(T *GLOBAL gptr)

Turns *gptr* into a local pointer (extracts the address part of a global pointer). Note that it is possible to dereference a global pointer without first turning it into a local pointer. On MANNA, the result is of type pointer to void.

T *GLOBAL MAKE_GPTR(int node, T *ptr)

Takes a node number and a local address and returns the corresponding global pointer. On MANNA, the result is of type pointer to void.

int OWNER_OF(T *GLOBAL gptr)

Returns the node pointed to by *gptr* (extracts the node part of a global pointer).

int IS_OWNER(T *GLOBAL gptr)

Returns *true* if *gptr* points to the local node.

Appendix B

Putting it all Together

The parallel execution of applications by the runtime system is reviewed in section B.1. Further, this section examines the RTS behavior in two specific cases - invoking a local function, and execution of a remote GET_SYNC_L operation. The Run-time system directory structure (as is presently on the EARTH SP-2 system) is presented in section B.4. Finally, using the portable Threaded-C system is mentioned in section B.5. The parallel execution of applications by the runtime system is reviewed in section B.1.

B.1 Parallel Execution

At any point of time only one application thread can run on a node. To achieve parallel execution in a NOW with distributed memory, multiple instances of the executable[1] are invoked on all the nodes[2] participating in the parallel execution. This means that the program variables are stored at the same relative positions within the application address space on each node, though there is no attempt to maintain coherence among their values across the nodes. The runtime system on these nodes supports inter-node communication through message-passing. The runtime system itself is coded in the SPMD programming model.

The application execution starts on node 0 with the first thread of the MAIN threaded function in the Threaded-C program. The `setjmp` statement in C is used to preserve the context before starting the application execution. As new tokens are generated on node 0, the load balancers on other nodes start sending load requests to the destinations

[1] Combination of object codes for the runtime system and Threaded-C application

[2] One instance per node.

chosen according to their balancer policy. When the rich nodes start responding to the load requests, the work load gets distributed across all the nodes in the parallel execution.

The execution stops when there are no more threads to execute. The runtime system on node 0 sends a termination message to all other nodes. Each node then executes the `longjmp` statement to revert back to their contexts before starting the parallel execution, and terminates.

B.2 Invoking a Local Function

Consider the Threaded-C code for invoking a local function in Fig. 2.3. The local function invocation statement, `INVOKE(NODE_ID, fib, SLOT_ADR(0), n-1, TO_GLOBAL(&r1))` is preprocessed into C code shown in Fig. 2.14. The sync slot and the result location address are converted into global addresses so as to be accessible by remote nodes (SLOT_ADR makes sync slot global, while TO_GLOBAL makes an address globally accessible). The parameter pointer is assigned the top of free element buffer on current node, and parameters assigned to the fields of this structure.

```
static void hdl_invoke (etc_handler fun,
                        long bytes)
{
  buf_elem *bp, *bn;

  bp = etc_rts.next_free;
  etc_rts.rdy_t->next = bp;
  etc_rts.rdy_t = bp;
  etc_rts.next_free = (bn= bp->next);
  bp->ip = fun;
  bp->fp = (long) bp;
  if (!bn) etc_alloc_buf_elem ();
  etc_get2n (bp->parms, bytes);
}
```

(a) Handler routine

```
void etc_invoke (int node, etc_handler fun,
                 long bytes)
{
 buf_elem *bp;

 if (node == etc_rts.node_id) {
   bp = etc_rts.next_free;
   etc_rts.rdy_t->next = bp;
   etc_rts.rdy_t = bp;
   etc_rts.next_free = bp->next;
   bp->ip = fun;
   bp->fp = (long) bp;
   if (!etc_rts.next_free)
       etc_alloc_buf_elem ();
 } else
     etc_send2n (node, hdl_invoke,
                 (int) fun, bytes,
                 etc_rts.next_free->parms,
                 bytes);
}
```

(b) RTS routine for INVOKE statement

Figure B.1: RTS performing Local Function Invocation

The last statement in the preprocessed code is `etc_invoke`, a function call to the RTS. The function `etc_invoke` is defined in the file `calls.c`. This function has the current node id (0 in this case), the instruction pointer for first thread in this threaded

function, and the size of the parameter structure stored on top of free element list in bytes, as arguments.

The function definition for `etc_invoke` is shown in part (b) of Fig. B.1. The node number (a parameter) is compared with the current node number. If they are the same, the top element from free element list is grabbed, it's instruction pointer filled and placed on the RQ. The free element list is a singly linked list of free nodes, each node with fields for instruction pointer, frame pointer, an array for parameters, and typical link fields (prev, next). Dynamic memory is usually accepted and returned from this list. Memory is allocated manually (by using the malloc statement), only when the free element list is empty. It may be noted that the top of the free element list already has parameters stored on it. On the contrary, if node number is not the same as current node number, then the arguments as well as the contents of top of free element list are composed into an active message with a handler routine to perform the invocation on remote node (though this case won't arise in case of local function invocation) and sent to neighboring node, the message destined to reach the node with id as the node number.

B.3 Execution of a Remote GET_SYNC_L

The implementation strategy of the EARTH communication primitives (GET_SYNC_X, and DATA_SYNC_X) had been mentioned in section 2.3.5. The behavior of a remote GET_SYNC_L is explained with an example in this section.

```
THREAD_1:
 time_1 = ct_read ();
 for (i = count; i > 0; i--) {
   GET_RSYNC_L (valp, resp, ssp0);
END_THREAD ();
 }
```

(a)Typical use of the GET_RSYNC_L primitive

```
_fp->time_1 = ct_read ();
if ((_fp->i = 100 * 1000),(_fp->i > 0))
 {
  etc_get_sync_l (_fp->valp, _fp->resp,
                  _fp->ssp0);
   return;
 }
```

(b)Preprocessed code for GET_RSYNC_L primitive

Figure B.2: Usage and preprocessed code for GET_RSYNC_L

Consider the use of the primitive GET_RSYNC_L as shown in Fig. B.2. The preprocessed code for this segment is shown in part (b) of Fig. B.2.

The GET_RSYNC_L primitive is used to request for `long` data from a remote location. As may be noticed in the generated code, a function call to RTS (etc_get_sync_l) is made. This function in the file `data.c` has a macro call in it's

body (`inline_etc_get_sync_l`) as shown in the Fig. B.3. The arguments for the macro are formed from the components of global pointer structures for source and destination locations. The macro definition for `inline_etc_get_sync_l` is included in the file `data_inc.c`. This macro checks if the source node number is the same as current node number, and if so, calls another handler routine (`inline_hdl_get_sync_l`), as shown in in Fig. B.5, that composes a `data_sync` message with the value taken from source location. The `data_sync` message places the value at the destination location and decrements the relevant sync count.

```
void etc_get_sync_l (etc_gptr src, etc_gptr dest,
                     etc_gptr s)
{
  inline_etc_get_sync_l (src.node, ((long *) src.ptr),
                         dest.node, ((long *) dest.ptr),
                   s.node, ((etc_slot *) s.ptr));
}
```

Figure B.3: RTS function etc_get_sync_l

```
static void hdl_get_sync_l (long *src_ptr, int dest_node,
                long *dest_ptr,int s_node, etc_slot *s_sp)
{
  inline_hdl_get_sync_l (src_ptr, dest_node, dest_ptr,
                         s_node, s_sp);
}
```

Figure B.4: Handler hdl_get_sync_l

On the other hand, if source node number is different from current node number, the `inline_etc_get_sync_l` routine sends an active message to the source node with a handler name `hdl_get_sync_l` in file `data.c`. This handler definition is shown in Fig. B.4. The handler routine `hdl_get_sync_l` makes a macro call to `inline_hdl_get_sync_call`, whose definition is in file `data_inc.c`. This macro, as mentioned above, composes a `data_sync` message to the destination location.

B.4 Run-Time System Directory

The portable EARTH programming environment consists of several tools, in particular the `etcc` compiler driver, the `etcpre` preprocessor and the run-time system libraries.

```
/*
void inline_hdl_get_sync_l (long *src_ptr,
   int dest_node, long *dest_ptr,int s_node,
   etc_slot_ptr s_sp)
*/
#define inline_hdl_get_sync_l(src_ptr,
   dest_node, dest_ptr, s_node, s_sp) \
{ \
inline_etc_data_sync_l ((*(long *) src_ptr),
     dest_node, dest_ptr, s_node, s_sp); \
}
```

(a) Macro inline_hdl_get_sync_l

```
#define inline_etc_get_sync_l(src_node,
        src_ptr, dest_node, dest_ptr,
        s_node,s_sp) \
{\
if (src_node == etc_rts.node_id) { \
    inline_hdl_get_sync_l (src_ptr,
    dest_node, dest_ptr, s_node, s_sp); \
}else{ \
   etc_send6 (src_node, hdl_get_sync_l,
   (int) src_ptr, dest_node, (int) dest_ptr,
   s_node, (int) s_sp, 0); \
   INC_REMOTE \
 } \
}
```

(b) Macro inline_etc_get_sync_l

Figure B.5: Macro Definitions in file `data_inc.c`

The EARTH home directory, pointed to by the EARTH_HOME environment variable, contains all these tools, as well as the necessary include files and libraries. The latest stable version of the source code is also available on some machines. The resulting directory structure for the portable runtime system is summarized in Fig. B.6.

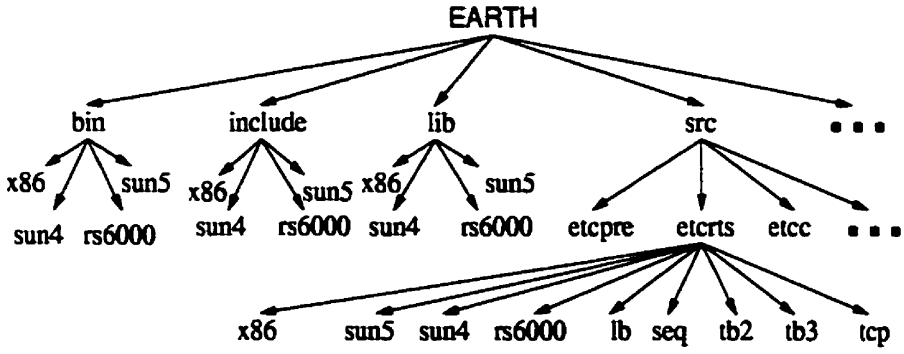


Figure B.6: Partial EARTH Directory Structure

As can be seen from Fig. B.6, three machine architectures are supported: Sun4/Sun5 (Sun workstations), rs6000 (RS6000 workstations and SP-2/SP-3 machines) and the Beowulf (LINUX PCs). In fact, MANNA would be a fourth architecture, but we do not include it in this diagram as the runtime system for the MANNA is written mostly in i860 assembly language, and therefore not portable across platforms. However, it may be noted that portable Threaded-C programs execute on the Manna architecture.

A closer look at the `etcrts` directory, which contains the source code for the run-time system libraries, shows separate subdirectories for the different machine architectures (`sun4`, `sun5`, `X86` and `rs6000`), the load balancers (`lb`), and each network implementation (`seq`, `tb2`, `tb3` and `TCP/IP`). Different combinations of the CPU and network interfaces may be specified on the command-line to enable linking of the application object code with different versions of the run-time system object code. The machine and network independent parts of the run-time system are linked with one of the machine-dependent modules, a network module to generate the final executable. In addition, one of the load balancers, and the profiling option may be included on the command-line at compile-time. Off-the-shelf NOWs may operate in sequential mode, or may be connected with the `TCP/IP` interface. NOW products like the IBM SP-2, SP-3 are interconnected with the `tb-2` and `tb-3` network interface cards. The command-line arguments are fed to the `etcc` compiler driver which after finding no errors, generates the final executable.

B.5 Running Threaded-C Programs

Presently, we support a 137 node IBM SP-2 at the Cornell Theory Center, an 80 node IBM SP-3 at the Argonne National Labs, and an 8 node Beowulf at the CAPSL Lab at the University of Delaware. This section details the compilation and execution sequences for Threaded-C programs on the IBM SP platforms.

In order to run Threaded-C programs, it is necessary to first compile them. In order for the Threaded-C compiler to work properly, the user's environment has to be set properly. The following paragraphs give a brief overview of what has to be done. More information can be found on the Web at `http://www-acaps.cs.mcgill.ca/info/EARTH`, though part of the information that is found there is specific to the EARTH-MANNA implementation.

In order to use the portable run-time system, the EARTH_HOME and the PATH environment variables have to be setup.

On the IBM SP-3 at the Argonne National Labs:

```
setenv EARTH_HOME ~kakulava/EARTH
```

On the IBM SP-2 at the Cornell Theory Center:

```
setenv EARTH_HOME ~kprasad/EARTH
```

The PATH variable should be set as follows:

```
set path=($EARTH_HOME/bin/rs6000 $path)
```

After the environment has been set up, programs can be compiled with the **etcc** command. This command is similar to the **cc** compiler driver and supports similar switches. For example, the following command can be used to compile the file `hello.c` and to generate a sequential version that runs on the Sun: **etcc** `-O hello.c -target sun4-seq`. The resulting Sun Sparc executable can be run as any sequential program. When accessing a parallel machine, on the other hand, some tool has to be used to get access to the machine and start the program on a specific number of nodes. The tools used for that purpose are : **tb3run** on the CACR SP-2, and **submit** on the Argonne SP-2 and Cornell SP-2. Note that this is not an exhaustive list, as more versions of the portable EARTH run-time system are being implemented.

A detailed explanation of the options for **etcc**, is present at `http://www-acaps.cs.mcgill.ca/info/EARTH/earth-manna/etcc.html`. The options supported by `etcc` are similar to those typically supported by other compilers. The relevant options for compiling Threaded-C programs on the IBM SP machines are as given below.

`etcc -target target-arch[-prof][-lb dual|spn|nop|snd|range|his|shis|rand|minima] [-h][v][-keep][-O[level]][-o file][-c][-S][-E] [-Dname[=def]][-Uname]` file ...

`-target target-arch` Specifies the target architecture. Code may be generated for the following IBM SP machines.

`rs6000-seq` : Generate a sequential executable that runs on a single IBM RS6000 CPU.

`rs6000-tb2` : Generate code for the Argonne SP-2 cluster of workstations. This implementation directly accesses the tb-2 network card for better performance.

`rs6000-tb3` : Generate code for the Caltech CACR IBM SP-2 cluster of workstations. This version also directly accesses the tb-3 network card.

`-prof` Link with a different version of the portable run-time system that gathers profiling information and prints it after the program has terminated.

`-lb dual|spn|nop|snd|range|his|shis|rand|minima` Selects one of the possible dynamic load balancers. the spn balancer is used by default, except for the sequential targets which use the nop load balancer.

For example, to run the program `fib.c`, the following steps are to be followed after setting the environment variable EARTH_HOME. Consider the platform to be IBM SP-3, load balancer to be dual, and the profiling option included.

```
etcc -target rs6000-tb3 -prof -lb dual -O2 -o
fib.dual.prof fib.c
```

Alternatively, the applications may be compiled using the `compile` script. Usage information is printed by typing `compile`.

compile [`cflag`] [`bal`] [`prof`] `progname`

- `cflag`: Any of the C optimization flags - O, O2,O3.
- `bal`: name of any of the load balancers, listed above.
- `prof`: Include profiling data in output.
- `progname`: Application name, without .c extension

To submit the executable for execution,

submit `num_nodes max_time program` [`arguments...`]

- `num_nodes`: Number of nodes requested.
- `max_time`: Maximum time that the application may take to execute.
- `program`: Name of the application exceutable
- `arguments`: Any application arguments may be included here

Usage information is printed by typing **submit**.

The resulting output gives the concatenated output of all nodes in one file. The profiling code if included, helps provide statistical information as to the number of threads generated, number of remote communications, number of tokens migrated etc.

Fig. B.7 shows a segment of the sample output for the Fibonacci Threaded-C program. The `fib` threaded function from Fig. 2.3 is compiled with the dual balancer and the profiling options. The executable is run with 32 nodes on the IBM SP-2 at the Cornell Theory Center. The sample output in Fig. B.7 shows the program result, and some profile information on node `0`. The actual output includes profiling information for various idle periods when the node is idle, and this sequence is repeated for all 32 nodes that participated in the execution. An explanation for the profile statistics is provided in Appendix C.

```
0:### Start (32 nodes, dual)
0:fib(33) = 5702887
0:ELAPSED Time is 31.467594 s
0:### Elapsed time: 31.468582 s
0:---------- PROFILING DATA for 0/32 ---------
0:Tokens consumed:  4317400
0:Tokens generated:  4894583
0:Threads running:  20040574
0:# of balancing activities:  1154366
0:# of requests sent:  0
0:# of requests received:  577183
0:# of tokens sent:  577183
0:# of tokens received:  0
0:# of remote communications: 808304
0:# of idle periods:  12
0:Total idle time:  1435.0414 (1433.6824) us
```

Figure B.7: Sample Output for the Fibonacci - Fib(33) on 32 nodes

Appendix C

Profiling support in the EARTH Runtime System

To monitor the performance of Threaded-C program execution, profile data is produced along with the application output on all the nodes participating in the parallel execution. This profiling support is based on a set of profiling parameters which account for the runtime system behavior during program execution. The generated profile data roughly falls into two categories[1]: a breakup of the total elapsed time with regard to the execution of application code and runtime system code; a count of the number of different individual RTS operations in the execution of a Threaded-C program. This profile data is useful in constructing a cost model for program execution on the EARTH RTS, and also identify possible design areas (both within the RTS and the application) for further optimization.

The profile code is mainly present in the files `prof.h` and `prof.c`. As may be noted from the command-line options of the `etcc` (EARTH Threaded-C Compiler), the `prof` option allows this profiling code to be linked in making the final executable. The profiling data is declared in `prof.h` and initialized in `prof.c`. Throughout the RTS, whenever a RTS operation is performed, relevant profile data is updated. The code to update this data is conditionally compiled along with the rest of the RTS (based on whether the `prof` option is included in the `etcc` command-line).

[1]The profiling support is currently implemented on the EARTH-SP2 and the EARTH-Beowulf. The breakup of the elapsed time is not currently available on the EARTH-Beowulf.

C.1 A Distribution of Total Elapsed Time

As the executable contains the object codes for both the Threaded-C application and the runtime system, it is important to study a breakup of the CPU time spent executing the application and implementing the runtime system functionality. Ideally the overheads for supporting a multi-threaded environment are expected to be minimal, and this cannot be overemphasized for fine-grain multi-threaded systems supporting non-blocking, non-preemptive threads. Therefore, a breakup of the total elapsed time offers a chance to make two important observations: application thread execution time versus time spent in the runtime system; and the relative comparison between times spent on different services offered by the runtime system. This comparison provides a detailed understanding of both the application and the runtime system behaviors during program execution. Inferences made from this study can be used to improve the application as well as the runtime system design.

The timing function used in Threaded-C programs is `etc_time()`. This function accesses the on-board timer and returns time in seconds with nanoseconds resolution[2]. The cost of executing the `etc_time()` function is 113 nanoseconds (14 cycles) . The minimum time that can be measured by this function is 238 nanoseconds (29 cycles). With this resolution, it might not be possible to measure correctly the time taken by some runtime system services which cost less than 29 cycles. However, this breakup is important in order to identify and categorize different runtime system services. Whenever a timer with better resolution is available in the future, those runtime system services which cannot be measured currently can be appended to provide a more realistic breakup of the elapsed time.

The elapsed time might be divided into seven components. Fig. C.1 shows a typical breakup of the CPU time on application and runtime system activities.

- *Application Execution Time*: Time spent executing application threads on a node.
- *Thread Management Time*: Time spent in spawning and supporting parallelism in the form of threaded functions and threads. The thread management functions that are considered here include parallel function invocation, filling thread specific data structures (frame pointer), and context-switching. Parallel function invocation involves the following items::

[2]On 120 MHz CPUs at the Cornell Theory center and the Argonne National labs (Quad), 1 cycle = 8.33 nanoseconds.

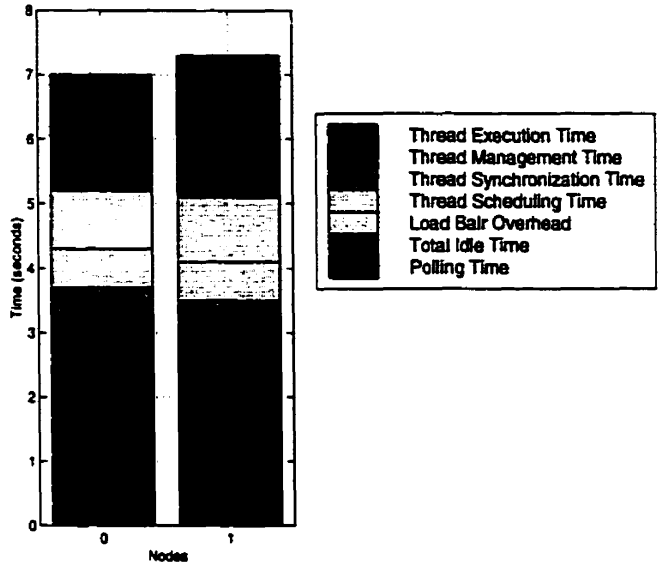


Figure C.1: A Breakup of Program Execution Time on 2 nodes - A Template

- Time spent in filling the parameter frame with arguments for child computations.
- Frame creation time.
- Frame filling time.
- Time spent in placing the child threaded functions in the ready queue or the token queue.

The term *context-switching* between two threads is the time from the exit of the first thread to the start of the second thread. Generally, context-switching between threads is a a light-weight operation than switching between processes. In systems that employ blocking multithreaded model, context-switching involves restoring the machine state and restarting the stalled thread. In EARTH, threads are non-blocking in nature, therefore when a thread exits there is no need to save the machine state. Context-switching here is as simple as an exit from a C function, and starting another C function. This time difference between these two events is the context-switching overhead between two threads. The total context-switching overhead on a node depends on the context-switching time between two threads and the total number of threads executed on a node.

Only context-switching overheads among the thread management functions above can be measured with the current timer resolution.

- *Thread Synchronization Time*:

  Thread synchronization time in EARTH is the time spent on satisfying control and data dependences between threads. There are no shared data structures because of two reasons: the memory model is based on distributed memory; and threads are non-blocking. Therefore the runtime system does not support any locks and this eliminates the synchronization overheads due to shared data access. The thread synchronization time in the runtime system consists of the the following components:

  - Time spent in preparing for data communication, i.e. time spent in handlers for sending data (this may include time spent in the network stack, until the data is placed in the final send buffers).
  - Time spent in invoking receiving handlers.
  - Time spent in local synchronizations (both communication-related and computation related). This time includes the time spent in executing the handler code and placing related threads in the RQ.

  Each occurrence of thread synchronization costs less than 29 cycles, and therefore is not instrumented in the runtime system.

- *Thread Scheduling Time*:

  The runtime system schedules an enabled thread at each thread boundary. An enabled thread is picked from one of the queues and dispatched for execution. The runtime system initially checks the ready queue for an enabled thread. If the ready queue is empty, it will schedule a thread from the token queue for execution. The thread scheduling time is the time spent at each thread boundary looking for a thread to start execution. Thread scheduling could not be measured with the current timer function.

- *Load Balancing Time*:

  Time spent in executing load balancing related handlers and functions. The handlers in the load balancer code access the two queues to add and remove tokens.

- *Idle Time*:

  The time spent by the CPU waiting for work to arrive. At the thread boundary, when the runtime system notices that both the ready and token queues are empty, it switches on the idle time counter and starts the load balancer. The idle time counter is switched off, when work arrives at the node, or work is generated on this node itself.

- *Polling Time*:

  The runtime system polls the network at every thread boundary. The network is also polled if a `POLL` statement is used in the Threaded-C program. This does not include the polling time initiated by the load balancer, which is masked under idle time.

C.2 Profile Data

A list of the items collected under the profile data is provided below.

- *Application Elapsed Time*: The total time for the execution of the application, from the start of first thread on node `0`, till the end of execution of last thread in the application.

- *Tokens Consumed*: The number of tokens consumed on this node.

- *Local Tokens Consumed*: The number of locally generated locally consumed tokens.

- *Tokens Generated*: The number of tokens generated on this node (by executing the function call `etc_token`).

- *Threads Running*: The number of threads executed on this node. It may be noted that the number of threads may not equal the number of tokens (the former is generally higher).

- *Number of Balancing Activities*: The total number of dynamic load balancing activities during the execution. This is the number of load balancing related communication activities (does not include the CPU time spent on load balancing).

No. of Load Balancing Activities = No. of Requests Sent + No. of Requests Received + No. of Tokens Sent + No. of Tokens Received

  - *Number of Requests Sent*: The number of requests for tokens sent by the load balancer on this node to remote nodes.
  - *Number of Requests Received*: The Number of requests received by this node from remote nodes.
  - *Number of Tokens Sent*: The number of tokens sent by the load balancer on this node to remote nodes (maybe as response to earlier received requests or because of the sender/hybrid policy of the load balancer).
  - *Number of Tokens Received*: The number of tokens received on this node (maybe as response to earlier sent requests, or because of sender/hybrid load balancer policies like `Snd`, `his`, etc. or maybe because this node is an intermediate node as part of the logical ring topology, while the token is on its way to destination).

- *Number of Remote Communications*: The total number of remote communications involved in implementing global memory management in remote memory access, synchronization, and spawning of threads `etc_spawn`. These are the communication activities spent in ensuring that certain RTS operations are performed at designated nodes, i.e. the intermediate message transfers until the message reaches the destination node (for the relevant handler to be invoked to implement the RTS operation). These communications don't include those required in implementing `INVOKE` and `TOKEN` and load balancing operations.

- *Extra Tokens*: Number of tokens received on a rich node. This count highlights the accuracy of the location policy of the balancer. It also helps focus on the accuracy of load state information and useless load-balancing during program execution.

- *Total Idle Time*: The total time that this node has been idle, i.e. without any application threads to execute. When a node has no threads to execute (both Ready Queue and Token Queue empty), it idles while polling the network.

- *Number of Idle Periods*: The number of idle periods throughout the application execution, that sum up to make the total idle time.

- *Idle Period Type*: The idle periods are of two types - application and balancer. A thread may be ready for execution in two ways - a token may be received as response for request sent earlier, or a token may be generated on this node itself. If the token is received as a result of load balancing, then the idle period is classified as *balancer* related idle period, whereas if the token is generated locally, the idle period is called *application* idle period.

- *Idle Period*: A time interval, indicating the start and end of each idle period. A set of such intervals are shown for all idle periods.

- *Balancer Idle Time*: Total balancer related idle time. It is a sum of all the time spent during balancer idle periods.

Appendix D

EARTH on Different IBM SP Installations

D.1 EARTH-SP at CACR, Caltech

The EARTH-SP2 (EARTH on RS/6000 IBM SP-2 system) at CACR[1], has 9 nodes (Power2 CPU) running at different individual clock speeds of 67MHz, 77MHz, and 135 MHz respectively. The L1 cache is of 32KB, L2 cache is 2 MB, and memory is of either 128MB or 512MB, depending on the node's classification. Network interface is through the tb-3 card. The point-to-point peak bandwidth on the 135MHz wide nodes is rated at 90.41MB/s and one way node-to-node latency is approximately 19 μs. The performance of EARTH on this platform is shown in Figures D.1, D.2, D.3 and D.4.

Operation	Local Seq.	Remote Seq.	Local Pipe.	Remote Pipe.
Sync Thread:	661 ns	20921 ns	178 ns	3147 ns
Spawn Thread:	639 ns	20888 ns	NA	NA
Read Word:	762 ns	41031 ns	241 ns	5563 ns
Write Word:	689 ns	41011 ns	201 ns	5642 ns
Fun. Call (1):	1445 ns	42165 ns	922 ns	6183 ns
Fun. Call (5):	1528 ns	43045 ns	986 ns	6879 ns
Fun. Call (9):	1631 ns	43141 ns	1060 ns	6917 ns
Fun. Call (18):	1686 ns	43225 ns	1154 ns	6527 ns

Table D.1: Overhead costs for EARTH operations on EARTH-SP2 (CACR)

[1] http://www.cacr.caltech.edu/resources/sp2/

EARTH Operation	Local Operation	Remote Operation	
	EU Costs	Local Costs	Remote Costs
SYNC	140 ns	2349 ns	896 ns
SPAWN	138 ns	2061 ns	1009 ns
END_THREAD	456 ns	NA	NA
INCR_SYNC	159 ns	1921 ns	898 ns
DATA_SYNC	163 ns	1920 ns	1127 ns
GET_SYNC	165 ns	1720+1685 ns	3143 ns
INVOKE(1)	117 ns	1901 ns	1101 ns
END_FUNCTION(1)	567 ns	NA	NA
INVOKE(5)	111 ns	1684 ns	1213 ns
END_FUNCTION(5	646 ns	NA	NA
INVOKE(9)	120 ns	1952 ns	1194 ns
END_FUNCTION(9)	691 ns	NA	NA
INVOKE(18)	152 ns	1842 ns	1265 ns
END_FUNCTION(18)	702 ns	NA	NA

Table D.2: Overhead for Threaded-C instructions on EARTH-SP2 (CACR)

Operation	Local		Remote	
	Overhead	Throughput	Overhead	Throughput
DATA_SYNC_B	249 ns/op	4.02 MB/s	5632 ns/op	0.18 MB/s
DATA_SYNC_S	240 ns/op	8.33 MB/s	5616 ns/op	0.36 MB/s
DATA_SYNC_L	228 ns/op	17.55 MB/s	5591 ns/op	0.72 MB/s
DATA_SYNC_D	212 ns/op	37.74 MB/s	5454 ns/op	1.47 MB/s
GET_SYNC_B	286 ns/op	3.50 MB/s	5615 ns/op	0.18 MB/s
GET_SYNC_S	287 ns/op	6.97 MB/s	5614 ns/op	0.36 MB/s
GET_SYNC_L	272 ns/op	14.72 MB/s	5620 ns/op	0.71 MB/s
GET_SYNC_D	296 ns/op	27.01 MB/s	5475 ns/op	1.46 MB/s

Table D.3: Overhead costs for GET_SYNC operation on EARTH-SP2 (CACR)

Align	Local		Remote	
	Single	Dual	Single	Dual
0	230.06 MB/s	230.99 MB/s	90.41 MB/s	98.04 MB/s
16	230.63 MB/s	230.61 MB/s	90.21 MB/s	97.61 MB/s
8	227.89 MB/s	229.67 MB/s	89.71 MB/s	96.19 MB/s
4	229.84 MB/s	230.11 MB/s	91.29 MB/s	97.44 MB/s
1	216.74 MB/s	217.38 MB/s	91.78 MB/s	97.35 MB/s

Table D.4: Bandwidth for Blockmove operations on EARTH-SP2 (CACR)

D.2 EARTH-SP at Argonne National Labs

D.2.1 IBM SP3 - Quad

The performance of the EARTH multithreaded environment on the IBM SP3 (Quad)[2] at the Argonne National Labs is reviewed here. The configuration of this platform is as follows: 80 node RS/6000 workstations, each single processor node running the P2SC CPU at 120 MHz. The tb-3 card is the network switch interface with a a peak bandwidth of 150 MB/s. Each node has 256 MB main memory, 256 KB cache and total disk space of 9 GB.

The latencies and overheads associated with EARTH operations are shown in Figures D.5, D.6, D.7 and D.8.

EARTH Operation	Local Operation	Remote Operation	
	EU Costs	Local Costs	Remote Costs
SYNC	104.41 ns	2526.86 ns	1723.87 ns
SPAWN	121.59 ns	2361.79 ns	2926.71 ns
END_THREAD	1205.91 ns	NA	NA
INCR_SYNC	175.96 ns	2362.96 ns	1730.98 ns
DATA_SYNC	145.84 ns	2324.96 ns	1834.14 ns
GET_SYNC	393.61 ns	2566.51+2229.79 ns	4491.60 ns
INVOKE(1)	129.12 ns	3229.34 ns	4033.40 ns
END_FUNCTION(1)	1353.34 ns	NA	NA
INVOKE(5)	121.88 ns	3040.48 ns	3786.50 ns
END_FUNCTION(5)	1421.65 ns	NA	NA
INVOKE(9)	139.06 ns	3374.81 ns	4158.61 ns
END_FUNCTION(9)	1485.84 ns	NA	NA
INVOKE(18)	157.78 ns	3825.99 ns	4730.58 ns
END_FUNCTION(18)	1529.01 ns	NA	NA

Table D.5: Overhead for Threaded-C instructions on EARTH-SP2 (Quad)

D.2.2 IBM SP2

This section documents the performance of the EARTH-SP2[3]. The performance figures shown in Figures D.9, D.10, D.11 and D.12, are based on the SP2 machine with 56 nodes

[2]http://www-fp.mcs.anl.gov/computing/machines/quad/

[3]Located at Argonne National Laboratories, Portland, USA

Operation	Local Seq.	Remote Seq.	Local Pipe.	Remote Pipe.
Sync Thread	1436.26 ns	24967.18 ns	200.37 ns	3914.76 ns
Spawn Thread	1404.437 ns	25354.698 ns	NA	NA
Read Word	1593.722 ns	47804.368 ns	267.105 ns	6863.099 ns
Write Word	1443.896 ns	47833.272 ns	239.688 ns	6614.358 ns
Fun. Call (1)	3236.203 ns	51388.782 ns	1735.278 ns	7882.220 ns
Fun. Call (5)	3086.590 ns	52467.688 ns	1825.649 ns	8404.783 ns
Fun. Call (9)	3179.877 ns	52709.572 ns	1902.766 ns	8733.468 ns
Fun. Call (18)	3247.824 ns	54818.314 ns	2001.446 ns	9029.014 ns

Table D.6: Overhead costs for EARTH operations on EARTH-SP2 (Quad)

Operation	Local		Remote	
	Overhead	Throughput	Overhead	Throughput
DATA_SYNC_B	271.00 ns/op	3.68 MB/s	6538.00 ns/op	0.15 MB/s
DATA_SYNC_S	273.00 ns/op	7.34 MB/s	6601.00 ns/op	0.30 MB/s
DATA_SYNC_L	260.00 ns/op	15.37 MB/s	6576.00 ns/op	0.61 MB/s
DATA_SYNC_D	241.00 ns/op	33.22 MB/s	6378.00 ns/op	1.25 MB/s
GET_SYNC_B	321.00 ns/op	3.11 MB/s	6831.00 ns/op	0.15 MB/s
GET_SYNC_S	324.00 ns/op	6.18 MB/s	6535.00 ns/op	0.31 MB/s
GET_SYNC_L	311.00 ns/op	12.85 MB/s	6512.00 ns/op	0.61 MB/s
GET_SYNC_D	339.00 ns/op	23.63 MB/s	6976.00 ns/op	1.15 MB/s

Table D.7: Overhead costs for GET_SYNC operation on EARTH-SP2 (Quad)

running at 62.5MHz, 128 MB memory, and an instruction/data cache of 32 KB each. The tb-2 card provides the network interface. Peak point-to-point bandwidth is 35 MB/s, and one way node-to-node latency is 30 microsecs. Access to this platform is no more

Align	Local		Remote	
	Single	Dual	Single	Dual
0	163.58 MB/s	166.40 MB/s	45.98 MB/s	47.97 MB/s
16	165.47 MB/s	166.27 MB/s	46.72 MB/s	47.75 MB/s
8	166.19 MB/s	166.56 MB/s	46.49 MB/s	48.16 MB/s
4	166.79 MB/s	163.92 MB/s	86.95 MB/s	90.22 MB/s
1	165.34 MB/s	166.44 MB/s	86.02 MB/s	88.97 MB/s

Table D.8: Bandwidth for Blockmove operations on EARTH-SP2 (Quad)

available.

The portable Threaded-C programs implemented to obtain the timing information of various EARTH operations are the same as those, described in ??. The performance is relatively better on the EARTH-SP2[4], than on the EARTH-SP2 reviewed here, because of faster clock speed, less network latency, and larger memory on the former, not withstanding the lack of homogeniety among it's nodes.

EARTH Operation	Local Operation	Remote Operation	
	EU Costs	Local Costs	Remote Costs
SYNC	301 ns	2486 ns	1895 ns
SPAWN	460 ns	2530 ns	2031 ns
END_THREAD	992 ns	NA	NA
INCR_SYNC	334 ns	2496 ns	1864 ns
DATA_SYNC	427 ns	2849 ns	2282 ns
GET_SYNC	550 ns	2664+2796 ns	4527 ns
INVOKE(1)	413 ns	3167 ns	2736 ns
END_FUNCTION(1)	1402 ns	NA	NA
INVOKE(5)	506 ns	2993 ns	2468 ns
END_FUNCTION(5)	1603 ns	NA	NA
INVOKE(9)	503 ns	2897 ns	2571 ns
END_FUNCTION(9)	1789 ns	NA	NA
INVOKE(18)	611 ns	3732 ns	2849 ns
END_FUNCTION(18)	1982 ns	NA	NA

Table D.9: Overhead for Threaded-C instructions on EARTH-SP2 (ANL)

[4]Located at CACR, Caltech, USA

Operation	Local Seq.	Remote Seq.	Local Pipe.	Remote Pipe.
Sync Thread	1773 ns	28699 ns	464 ns	4047 ns
Spawn Thread	1561 ns	28595 ns	NA	NA
Read Word	2156 ns	56742 ns	690 ns	7249 ns
Write Word	1934 ns	56121 ns	609 ns	7077 ns
Fun. Call (1)	3863 ns	60094 ns	2365 ns	8364 ns
Fun. Call (5)	4102 ns	60713 ns	2603 ns	8593 ns
Fun. Call (9)	4354 ns	61077 ns	2863 ns	8761 ns
Fun. Call (18)	4853 ns	63651 ns	3325 ns	9659 ns

Table D.10: Overhead costs for EARTH operations on EARTH-SP2 (ANL)

Operation	Local		Remote	
	Overhead	Throughput	Overhead	Throughput
DATA_SYNC_B	810 ns/op	1.24 MB/s	7071 ns/op	0.14 MB/s
DATA_SYNC_S	817 ns/op	2.45 MB/s	7101 ns/op	0.28 MB/s
DATA_SYNC_L	811 ns/op	4.93 MB/s	7066 ns/op	0.57 MB/s
DATA_SYNC_D	918 ns/op	8.71 MB/s	7156 ns/op	1.12 MB/s
GET_SYNC_B	890 ns/op	1.12 MB/s	7095 ns/op	0.14 MB/s
GET_SYNC_S	896 ns/op	2.23 MB/s	7081 ns/op	0.28 MB/s
GET_SYNC_L	908 ns/op	4.40 MB/s	7088 ns/op	0.56 MB/s
GET_SYNC_D	967 ns/op	8.28 MB/s	7159 ns/op	1.12 MB/s

Table D.11: Overhead costs for GET_SYNC operation on EARTH-SP2 (ANL)

Align	Local		Remote	
	Single	Dual	Single	Dual
0	69.96 MB/s	69.93 MB/s	34.09 MB/s	39.46 MB/s
16	69.83 MB/s	69.88 MB/s	34.09 MB/s	39.23 MB/s
8	69.83 MB/s	69.75 MB/s	34.12 MB/s	39.72 MB/s
4	71.16 MB/s	71.16 MB/s	32.90 MB/s	33.69 MB/s
1	66.32 MB/s	66.29 MB/s	32.65 MB/s	33.65 MB/s

Table D.12: Bandwidth for Blockmove operations on EARTH-SP2 (ANL)

Appendix E

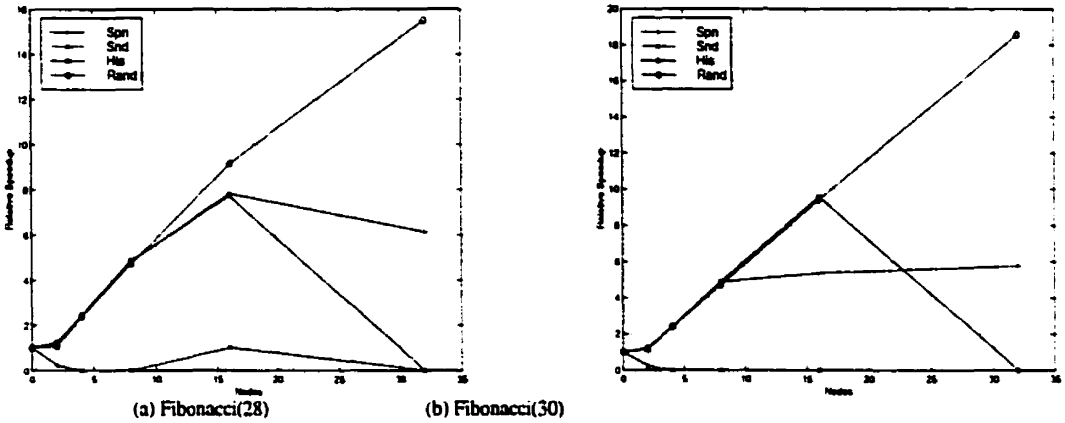
Additional Experiments

Benchmark	Dual	Spn	Shis	Snd	His	Range	Catapult	Rand
Fibonacci(33)	0.807	0.86	9.82	26	OF	0.93	0.92	0.809
Queens(12)	0.28	0.17	4.68	5.04	0.18	0.2	OF	0.23
TSP(10)	0.4	0.28	7.76	8.65	0.29	0.29	0.28	0.32
Knary(7, 7,2)	2.13	0.93	24.76	1.037	0.91	0.94	0.95	0.91
Knary(2,512,0)	0.053	0.013	0.17	0.171	0.0067	0.013	NA	-
Matrix(1024X1024)	70.31	49.53	293.79	17.52	12.21	14.66	63.42	16.96
Tomcatv(257)	2.45	1.78	OF	OF	0.54	OF	OF	5.6
SPMD(1,1,0)	0.25	0.16	0.68	0.08	0.11	0.1	0.63	0.15
SPMD(4,4,0)	1.9	0.72	14	0.63	0.86	1.27	13	0.79
Paraffins(28)	7.42	6.69	104.1	123.3	6.72	6.75	NA	6.65

Table E.1: Overview of Results. Elapsed times in **seconds** are shown for different benchmarks belonging to the recursive (divide-and-conquer), regular and irregular programming models against various dynamic load balancers belonging to the receiver-initiated, sender-initiated and hybrid categories. Measurements are based on **32 node** runs. The elapsed times do not include profiling overheads. Table 5.1 shows elapsed times with profiling effects

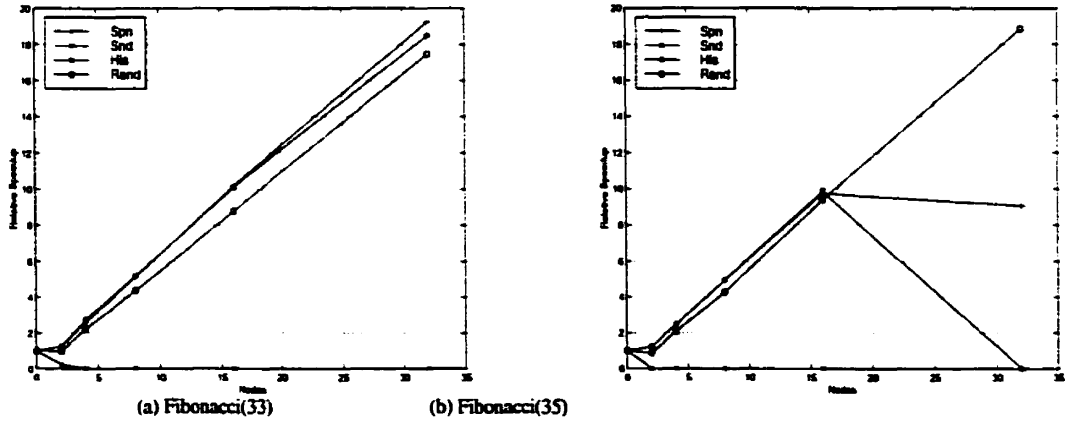
Benchmark	Dual	Spn	Shis	Snd	His	Range	Catapult	Rand
Fibonacci(33)	1	3	6	7	8	5	4	2
Queens(12)	5	1	6	7	2	4	8	3
TSP(10)	6	1	7	8	4	3	2	5
Knary(7, 7,2)	6	4	7	8	1	3	2	5
Knary(2,512,0)	5	2	6	7	1	3	8	4
Matrix(1024X1024)	8	3	5	8	8	1	4	2
Tomcatv(257)	4	3	8	8	2	1	8	5
SPMD(1,1,0)	5	3	7	8	1	4	6	2
SPMD(4,4,0)	5	1	8	7	3	4	6	2
Paraffins(28)	5	2	6	7	3	4	8	1
Average	5	2.3	6.6	7.5	3.3	3.2	5.9	2.8
Rank	5	1	7	8	4	3	6	2

Table E.2: Relative ranking of the different balancers based on their performance as shown in Table E.1



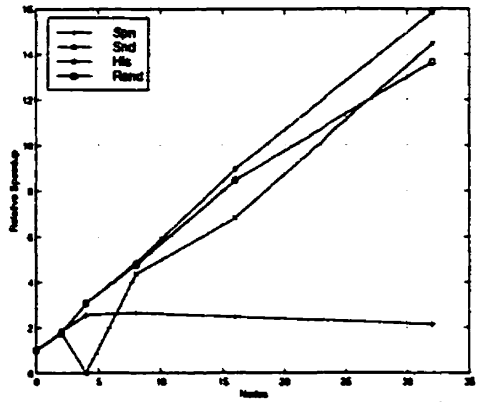
(a) Fibonacci(28) (b) Fibonacci(30)

Figure E.1: Relative Speedups of different balancers for Fibonacci

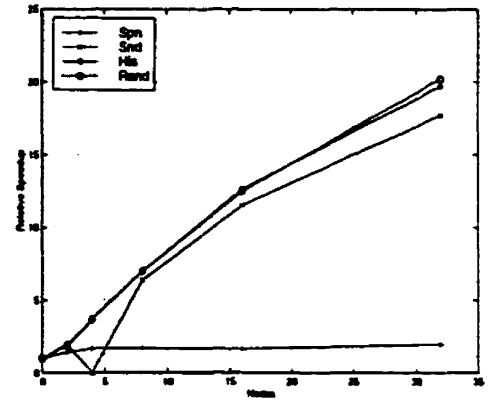


(a) Fibonacci(33) (b) Fibonacci(35)

Figure E.2: Relative Speedups of different balancers for Fibonacci

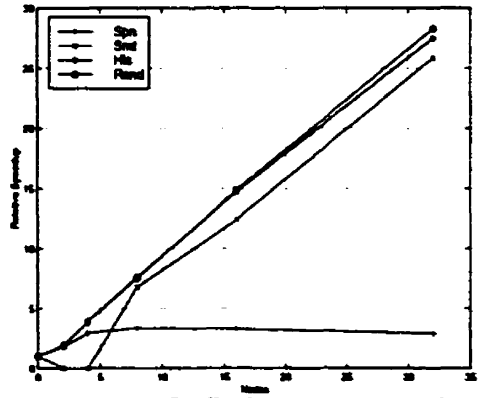


(a) Paraffins(22)

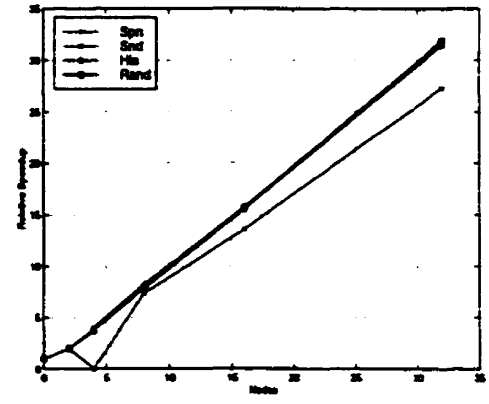


(b) Paraffins(24)

Figure E.3: Relative Speedups of different balancers for Paraffins



(a) Paraffins(26)



(b) Paraffins(28)

Figure E.4: Relative Speedups of different balancers for Paraffins