Dynamic Load Balancers for a Multithreaded Multiprocessor System

Prasad Kakulavarapu* Olivier C. Maquelin* José Nelson Amaral†
Guang R. Gao‡

ABSTRACT

Designing multi-processor systems that deliver a reasonable price-performance ratio using off-the-shelf processor and compiler technologies is a major challenge. For an important class of applications, it is critical to explore fine-grain parallelism to achieve reasonable performance. In such parallel systems it is essential to efficiently manage communication latencies, bandwidth, and synchronization overheads. In this paper we study load balancing strategies for the runtime system of a multi-threaded system. EARTH (Efficient Architecture for Running Threads) is a multi-threaded programming and execution model that supports fine-grain, non-preemptive, threads in a distributed memory environment. We describe the design and implementation of a set of dynamic load balancing algorithms, and study their performance in divide-and-conquer, regular, and irregular applications. Our experimental study on the distributed memory multiprocessor IBM SP-2 indicate that a randomized load balancer perform as well as, and often better than, history based load balancers.

1 Introduction

Multithreading allows the effective management of communication latencies and the efficient implementation of synchronizations in parallel computing and enables the exploration of fine-grain parallelism [10,19,26]. 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 parallelism in irregular applications. Fine-grain parallelism, on the other hand, allows higher processor utilization, and thus is more suited for applications that have irregular data accesses and a dynamic distribution of workload. However most fine-grain parallel systems are constrained by communication overheads. This article reports an implementation of a multithreading system that demonstrates the viability of implementing multithreaded models with existing off-the-shelf distributed memory parallel systems.

According to a recent study, in order to support fine-grain treads efficiently at runtime, threads should be *abundant*, *balanced*, and *cheap* [22]. Having an abundant number of active threads on a multi-processor system increases processor utilization, because if one

*This paper is the result of the author's work prior to their employment at Intel Corporation, and does not represent the positions or products of Intel Corporation in any way. Email: {prasad.kakulavarapu, olivier.maquelin}@intel.com

†Dept. of Computing Science, University of Alberta, Edmonton, AB, Canada, Email: amaral@cs.ualberta.ca

‡Dept. of Electrical and Computer Engineering, CAPSL Lab, University of Delaware, Newark, USA, Email: ggao@capsl.udel.edu

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 to enable this operations to take place frequently.

The Efficient Architecture for Running THreads (EARTH) is a multi-threaded architecture and execution model that supports fine-grain, non-preemptive (and non-blocking) threads. EARTH allows the implementation of a multi-threaded execution model with off-the-shelf microprocessors in a distributed memory environment [10]. In order to reduce delays incurred because of transfers between the application level and the operating system level, 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.

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.

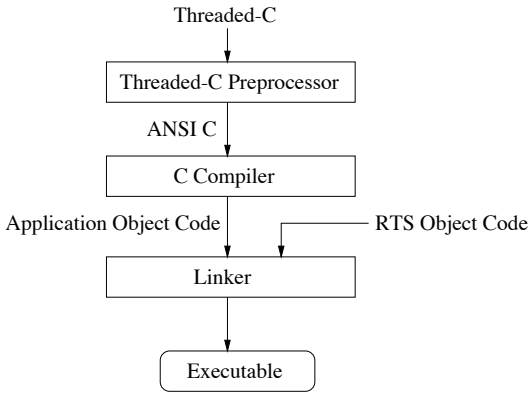


Fig. 1: Translation Sequence of Threaded-C code

The translation sequence for programs written in Threaded-C is shown in Fig. 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 final executable is obtained by linking the application object code with the runtime system object code.

EARTH is currently implemented on multiple platforms - network of Sun workstations, MANNA [10], IBM SP-2 [5], Beowulf [13], and a SUN SMP cluster. All platforms except for the Sun SMP cluster are distributed memory implementations. Earlier studies on EARTH [10] described the implementation of the EARTH model on the MANNA ma-

chine, which has two processors in each processing node. In this article we report results of the EARTH implementation on the IBM SP-2 where activities of the EARTH model are supported by a single processor in each node.

In this paper we present the design of a new randomizing load balancer that performs well for irregular or recursive workloads. We also report experimental results that allow the comparison of the new load balancing strategy with a set of existing EARTH load balancers. In the next section we describe the EARTH multithreading system and its programming model. In Section 3 we briefly describe the dynamic load balancers implemented in the EARTH runtime system. Section 4 describes the experimental framework used to study the performance of these load balancers. The actual performance results and its analysis are presented in Section 5. We discuss related work in Section 6 and present our conclusions in Section 7.

2 The EARTH Multithreading System

Applications for the EARTH architecture are written in Threaded-C [23], a multi-threaded variant of C. Threaded-C can also be used as a compilation target for other parallel languages [9]. Threaded-C provides constructs for the definition of fine grain, non-preemptive, non-blocking 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.

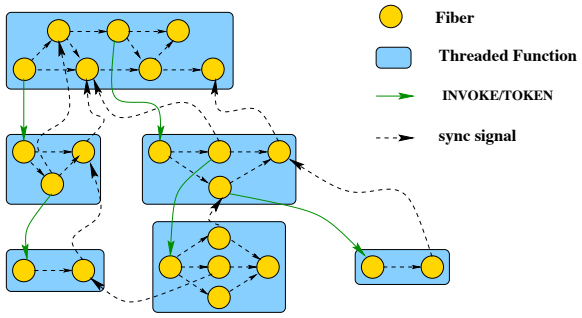


Fig. 2: Arbitrary activation graph allowed by the use of synchronization slots.

A *thread* is a set of instructions that is executed to completion without suspension or blocking. To enable context sharing among related threads, these threads are grouped into larger units called *threaded functions* [23]. A threaded function can allocate an array of synchronization 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. Whenever a synchronization signal for a slot is received, the slot counter is decremented. 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. The sync slots mechanism provides a unique handle to signal individual threads and enables the definition of any arbitrary thread activation graph. Figure 2 illustrates the arbitrary activation graphs that can be constructed using the sync slot

mechanism. In this figure, a rectangular block represents a threaded function, and a circle represent a thread. Parallel function calls are shown as solid arcs while dashed arcs between 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. Figure 3 shows the activation tree for a recursive implementation of the Fibonacci function.

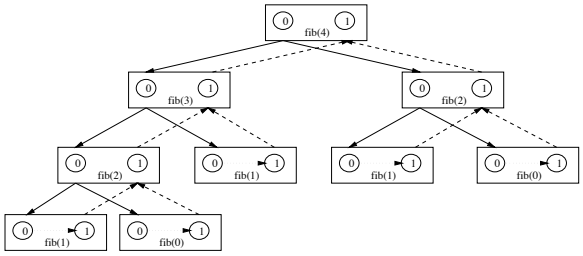


Fig. 3: Activation Tree for Fib(4) and a generic activation graph for a Threaded-C program.

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 is given in [23].

2.1 The EARTH Runtime System

The EARTH *runtime system (RTS)* provides a multi-threaded environment for running multi-threaded programs efficiently. The core responsibilities of the RTS includes thread-scheduling, context switching, data communication, synchronization, global memory management, and dynamic load balancing. The services provided by the RTS that require inter-node data communication are based on the technique of active messages [27]. 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. For efficiency, and to isolate the interactions between the network and the rest of the RTS, a limited set of functions is used for inter-node communication.

2.2 Thread Scheduling

There are two Threaded-C primitives that allow the creation of a threaded function activation. One of this primitives, `INVOKE`, specifies the processor that will execute the activation while the other, `TOKEN`, leave the determination of this processor for the runtime system. The execution of the `TOKEN` primitive by a running thread causes the creation of a *token*. This token is formed by a unique identifier of the threaded function to be executed and by a set of values for the parameters of the function. Such tokens are the unit of load balancing in the EARTH system.

The EARTH runtime system maintains two queues — the `ready queue (RQ)` and the `token queue (TQ)` — in each processing node. The RQ contains enabled *threads*, while the TQ contains *tokens*. A thread is associated with an *activation frame* that is a data structure containing the values of local variables and parameters associated with a

threaded function activation. Typically the consumption of a token by a processing node will lead, eventually, to the enabling of multiple threads. Each of these threads, when enabled, will run to completion without blocking. However because the threaded function is formed by multiple threads, the execution of such a function can and will block to wait for synchronization signals.

Whenever a processor runs out of enabled threads in its RQ, it will fetch a token from its TQ for execution. This process requires the creation of an activation frame for the threaded function and the placement of the thread `0` of the function in the RQ.[1] Once the token is transformed in the actual function activation, it can no longer migrate to other processors. As a consequence all the threads within a threaded function must execute on the same processing node [10,15]. Notice that at any given time threads from multiple functions can be enabled and dormant in a given processor node. Moreover if all threads are dormant, the RQ will be empty, and the runtime system will fetch a token from the TQ.

The TQ is a `DEQUE` - a data structure similar to a queue, but operable on both ends. The TQ behaves locally like a stack to the local node and like a FIFO to the load balancer that is extracting tokens to send to other nodes. 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, tokens are extracted from the head of the TQ for remote consumption. When tokens are received from remote nodes they are added to the head of the TQ. The flow of tokens amongst the application, RQ, TQ and remote nodes is shown in Fig. 4.

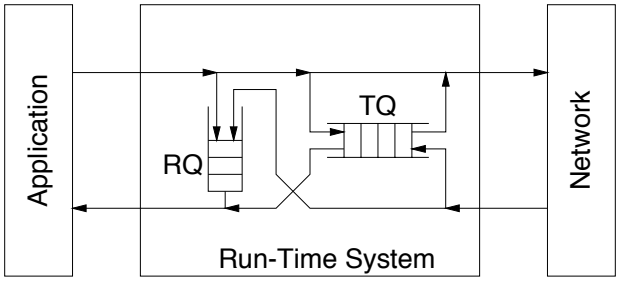


Fig. 4: Internal Queues in the EARTH Runtime System

Maintaining separate queues for migratable threaded functions and threads bound to the local processing node reduces the amount of memory consumed to store the activation frames of dormant threaded functions. The TQ design favors the local execution of tokens produced locally. It also favors the local execution of the tokens produced most recently. This policy exploits both spatial and temporal locality in the reference to data structures shared among threaded functions. The policies selected to manipulate the TQ and the RQ affect the order of execution of the parallel functions and might determine the amount of parallelism that can be exploited and the amount of memory needed to execute the program [15].

The migration of functions to other nodes depends on (1) the number of nodes in the multi-processor system, (2) the time needed to start a threaded function on another node, (3) the amount of work available to the other nodes, and (4) the efficiency of the dynamic

[1] The thread `0` of a threaded function is a special thread that is enabled only when the function is first activated and is used for initialization.

load balancing mechanism. For instance, a depth-first expansion of a recursive activation tree should take place locally, whereas breadth-first expansion should be distributed over a set of nodes. Function frames at higher levels in the activation tree are likely to 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 migration costs with the work done on the remote node. Depth-first expansion on the local node not only reduces token migrations, but also adds to the locality of the tokens migrated.

3 Dynamic load Balancing in EARTH

A typical load balancer algorithm has four phases - processor load evaluation, load balancing profitability determination, task selection, and task migration [21]. These phases requires the establishment of (1) a *transfer policy* to determine if load should be transferred to another node; (2) a *task selection policy* to identify a good candidate for migration; (3) a *destination selection policy* to chose a destination to the task that must migrate; and a (4) *information collection policy* to decide what load information should be collected, how often it should be collected and where it should be stored.

The load balancing goal in EARTH is *to ensure that all nodes are busy* rather than to evenly distribute tokens among nodes. A node is *poor* when it has no threads to execute and no tokens in its token queue. A node that has more then a fixed number of tokens in its token queue is a rich node. 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.

Balancers can be *receiver-initiated* (work-stealing), *sender-initiated* (work-sharing), or *hybrid* (symmetric) [21,5]. In receiver-initiated load balancers the overhead of balancing the load is incurred mostly by poor nodes. Because the load balancing actions are triggered by change in local state, this approach results in minimum overheads. Sender-initiated load balancer typically either incur extra overhead for the search for a poor node, or for the retransmission of load when the selected destination node is rich. Hybrid balancers combine a sender and a receiver component to benefit from the advantages from both receiver-initiated and sender-initiated balancers.

The goal in the EARTH runtime system is to design simple balancers that deliver good load distribution with minimum overheads. Two types of load balancing messages are exchanged among the nodes: (a) a *load balancing request* is typically served immediately if the receiving node has surplus tokens, but in some balancers the message might be forwarded to another node; a *response to load balancing request* contains a token from a remote node to be processed in the receiving node. A virtual ring network topology is adopted in all the balancers with nodes numbered clock-wise. We present a summary description of the existing load balancing policies in EARTH. For a detailed description of these policies see [11,5].

Dual: Requests circulate counter-clockwise in the ring, and tokens circulate clockwise. An idle node generates a request into the ring. When the request reaches a rich node, that node sends a token back.

Spn: To reduce traffic, a request contains the identification of the node that originated it. The reply is sent directly to the requester.

Shis: The node that replies to a request attaches its own id to the token. The node that gets the token “remembers” which node is rich and next time sends a request directly to that

node. If that fails the request is forwarded along the ring as in *Dual*.
Snd: When the TQ of a rich node surpass a threshold, it sends a token around the ring. Poor nodes grab tokens but do not generate requests. If tokens arrive at their generators, they are taken out of the ring.
His: Each node monitors load balancing messages and builds a list of nodes that are likely to be poor and rich. If the node becomes poor, it sends a request to the richest node in the list. Nodes that become reach send tokens to poor nodes.
Range: Similar to Dual and Spn, but 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. When the node is rich, tokens are sent to the nodes in the far end the range.
Catapult: Similar to Range, but the tokens are sent to the near end of the range list.

3.1 The Rand *Balancer*

The *Rand* balancer is a hybrid randomized load balancer inspired in the supermarket model proposed for distributed computing [18]. In the receiver mode, load probes are sent to randomly chosen nodes. The least loaded node is chosen as the destination node for a load transfer. The balancer assumes a *completely connected graph* as a logical topology between the nodes, *i.e.*, messages are sent directly from one node to another without intervention from the CPU of intermediate nodes. A load threshold is used to limit excessive load transfers in the sender mode.

A node should send enough load probes to have a reasonable expectation of finding a poor node, but the load probes should not overload the network. After much experimentation we decided to use $d = (Number\ of\ Nodes)/10 + 1$ load probes. We keep the value of d constant. This number of probes 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.

The sender mode of the *Rand* balancer is switched on when the number of entries in the local token queue is greater than twice the number of probes sent. The number of probes sent, in turn, depends on the number of nodes in the execution. The receiver mode of the *Rand* balancer is switched on when the node is idle. A simple task selection policy is used: the token on the top of the token queue is always selected for migration. This token is expected to be higher in the activation tree of recursive functions, and thus should carry more work to the receiving node.

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. When a load request or load probe is received, we assume that the sender has zero workload. Local load information is also piggy-backed on outgoing tokens.

4 Experimental Framework

The Threaded-C programs used in our experiments can be classified according to their data access and workload generation pattern as divide-and-conquer, regular, and irregular algorithms. Most of these programs produce 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 it is critical to minimize the load balancing overheads [5].

Benchmark Name	Problem Domain	Type	Tokens Generated	Threads executed
`Fibonacci` (33)	Combinatorial	Divide and conquer	11405772	17108661
`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`	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 1: The EARTH Benchmark Suite

The benchmark programs used in our experiments, shown in Table 1, are taken from the EARTH Benchmark Suite (EBS) [25]. The table shows the number of tokens generated and the number of threads executed for each benchmark. `Fibonacci`, `Queen` and the Traveling Salesperson Problem `TSP` are typical examples of recursive divide-and-conquer algorithms. `Paraffins` generates irregular workload and produces tokens with short execution time. `Matrix` (standard matrix multiplication) and `Tomcatv` are regular scientific computations, while `SPMD` models a typical barrier-synchronized application.

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 [1]. 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 descriptions of the benchmarks, experimental platform, and all our experiments can be found in [11].

5 Performance Results

We used the benchmarks described in Section 4 to evaluate the load balancers described in Section 3 in the IBM SP2 machine at the Cornell Theory Center. We conducted extensive experimentations to evaluate the load balancers. In this section we are summarizing the most important results.

5.1 Summary of Main Results

Some of the most important results from our experiments can be summarized as follows.

- Dynamic load balancing is most effective for programs that produce am abundant number of migratable threads. Abundance of threads is even more important for irregular or highly recursive programs.
- The *His*, *Range*, and *Rand* balancers perform very well for most applications. For highly recursive or irregular programs the best results were obtained by the *Rand* balancer (see Table 2).
- The history based *His*, and the *Range* balancers perform very well in `SPMD` style programs where the load distribution is quite regular and predictable (see Table 3).

- Dynamic load balancing is essential for good performance. For most applications the speedup over the *Minima* balancer — which shuts itself off after an initial (thus static) distribution of load — was above 18 for execution in 32 nodes (see Table 4).
- The overhead for supporting multi-threading in the EARTH model is quite low (except for extreme cases such as the recursive `Fibonacci` — see Table 5).

In order for dynamic load balancing to be successful, some basic conditions must exits: (1) the application program must have enough parallelism to allow it to be broken down into a large enough number of small threads to allow the load balancers to work; (2) the benefit from migrating work to other processors must surpass the cost of executing the load balancer algorithm itself. Some load balance strategies also depend on the regularity and predictability of the work load generated across the machine. In this study we analyze how well the load balancers described in Section 3 perform as these conditions change.

5.2 Comparison of the Balancers

Benchmark	Dual	Spn	Shis	Snd	His	Range	Catapult	Rand
`Fibonacci`(33)	1.14	1.14	13.66	—	1.19	1.21	1.2	1.02
`Queens`(12)	0.24	0.17	4.71	0.171	0.176	0.175	—	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
`Matrix`	70.31	49.5	293.8	17.52	12.2	14.66	63.42	16.96
`Tomcatv`(257)	2.45	1.78	—	—	0.54	0.39	—	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	—	6.46

Table 2: Overview of Results. Elapsed times, in seconds, for each benchmark under different load balancing algorithms. These measurements were obtained in a 32 node run.

Table 2 shows the elapsed times, measured in seconds, for different balancers and threaded programs. The entries missing in the table are cases in which the load balancer generates so much network traffic at some point during the execution of the program that it saturates the network interface buffer. In the implementation of these benchmarks, there is no throttling to prevent this situation. The same data is normalized into Table 3. For normalization we divide each elapsed time by the shortest elapsed time of any balancer for that program. Thus, Table 3 actually shows the relative slowdown of each load balancer in relation to the fastest one. To compute a fair average for each application program group, we assign the longest elapsed time of any of the load balancers that successfully complete the execution of the program to the load balancers that fail to complete them. In this table `D&C Average` is the average for the divide-and-conquer applications, and `Reg. Average` is the average for regular applications.

From the data in Tables 2 and 3 we can conclude that although the performances of the *His*, *Range*, and *Rand* balancers are quite similar, the *Rand* balancer performs better for divide-and-conquer programs (the top four programs in the table), and in paraffins, the only program with irregular data distribution and irregular workload generation in our test set. As should be expected, load balancers that use load history perform well when executing programs with regular data distribution and load generation.

Benchmark	Dual	Spn	Shis	Snd	His	Range	Catapult	Rand
`Fibonacci`(33)	1.12	1.12	13.4	—	1.17	1.19	1.18	1.00
`Queens`(12)	1.45	1.03	28.5	1.04	1.07	1.06	—	1.00
`TSP`(10)	1.59	1.19	28.9	1.33	1.04	1.07	1.04	1.00
`Knary`(7, 7,2)	2.25	1.03	27.3	1.14	1.00	1.04	1.05	1.00
D&C Average	**1.63**	**1.09**	**24.5**	**4.23**	**1.07**	**1.09**	**7.95**	**1.00**
`Matrix`	5.76	4.06	24.1	1.44	1.00	1.20	5.20	1.39
`Tomcatv`(257)	6.28	4.56	—	—	1.38	1.00	—	14.4
`SPMD`(1,1,0)	3.13	2.00	8.50	1.00	1.38	1.25	7.88	1.88
`SPMD`(4,4,0)	3.02	1.14	22.2	1.00	1.37	2.02	20.6	1.25
Reg. Average	**4.55**	**2.94**	**17.3**	**4.45**	**1.28**	**1.37**	**12.0**	**4.72**
`Paraffins`(28)	1.15	1.01	16.1	1.17	1.01	1.05	—	**1.00**

Table 3: Normalized elapsed times, and averages for each class of programs and for the entire set of tests.

In both instances of `SPMD`, a barrier-synchronizing program, the *Snd* balancer outperforms all other balancers. Typically in these applications the number of tokens generated between consecutive barriers is not much higher than the number of nodes in the system. Therefore the load balancer does not have many tokens to exchange to balance the load. *Snd* does well with these programs because it is quite fast on distributing newly generated tokens. However the current implementation of the *Snd* balancer might overflow the network buffers in applications that generate an abundant amount of tokens.

The *Range* and *Spn* are very similar balances and their performance reflects that. The sender component of the *Range* balancer sends extra tokens to the far node in the range list. This policy 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. The *Catapult* location policy, *i.e.*, the identification of the node that should receive extra load is misguided. By always choosing the nearest neighbor to send extra tokens, the Catapult balancer is likely to shift excess work to another processor instead of distributing it among a range of idle processors.

The ring topology assumed in the *Dual* balancer severely limits its scalability, its ability to respond rapidly to fluctuating load situations, and its capability to distribute tokens. Nonetheless, its simple algorithm makes it useful for applications with threads that run for a short amount of time. The *Shis* balancer causes high message traffic, resulting in poor performance in spite of the use of history information to select nodes to distribute the load. The poor performance of the *His* balancer underscores an important principle of load balancer design: a good and fast decision is better than an optimal but slower one.

5.3 Dynamic × Static Load Balancing

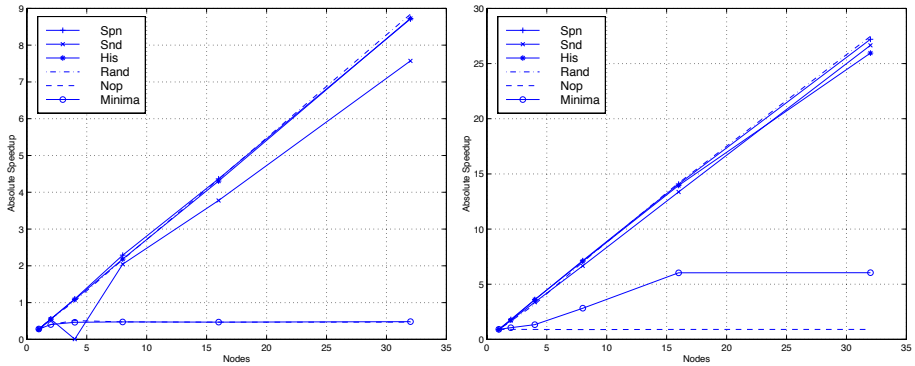
In Tables 2 and 3 we compare the performance of the load balancers amongst themselves. But how important is it that the load be balanced dynamically? Could an initial static distribution of the load produce similar results? To answer these questions we run the same set of programs under a minimal balancing policy. The *Minima* balancer shuts itself off after initially distributing one token to each node. The results of these experiments are presented in Table 4. In a run that uses 32 processing nodes, the dynamic balancers produced a speedup over the static load distribution (Minima) that ranges from 4.30 for `Queens`(12)

Benchmark	Attribute	Spn	Snd	His	Rand	Minima
`Fibonacci`(33)	Elapsed Time	1.14	OF	1.19	1.02	23.29
	% Reduction	95.12	–	94.91	95.63	0
	Speedup	20.50	–	19.65	22.89	1
`Queens`(12)	Elapsed Time	0.167	0.171	0.176	0.166	0.754
	% Reduction	77.79	77.34	76.72	78.01	0
	Speedup	4.50	4.41	4.30	4.55	1
`TSP`(10)	Elapsed Time	0.32	0.36	0.275	0.269	7.78
	% Reduction	95.91	95.34	96.47	96.54	0
	Speedup	24.45	21.45	28.35	28.90	1
`Knary`(7,7,2)	Elapsed Time	0.93	1.04	0.907	0.906	24.77
	% Reduction	96.26	95.81	96.34	96.34	0
	Speedup	26.72	23.88	27.30	27.33	1
`SPMD`(4,4,0)	Elapsed Time	0.72	0.63	0.86	0.79	14.037
	% Reduction	94.87	95.50	93.88	94.34	0
	Speedup	19.50	22.21	16.34	17.67	1
`Paraffins`(28)	Elapsed Time	6.55	7.54	6.54	6.46	118.8
	% Reduction	94.48	93.65	94.49	94.56	0
	Speedup	18.13	15.76	18.16	18.40	1

Table 4: Comparison with a no balancer *Minima* run in a 32 processor machine. Elapsed time is measured in seconds.

with the His balancer, to 28.9 for `TSP`(10) with the Rand balancer. Typically the speedup is around 20 for 32 processors. Our implementation of `Queens`(12) is throttled, therefore we may consider that the programmer is doing part of the load balancing, and therefore the dynamic load balancing is not as important for that program. The results presented in Table 4 is a remarkable and strong indication of the importance of a good dynamic load balancer to deliver high performance in a programming model such as EARTH.

5.4 Speedup Curves



(a) `Paraffins`(28) (b) `Queens`(12)

Fig. 5: Performance comparison between Minima, Nop and other Balancers of different balancers

In order to understand how the speed is increasing with the number of processors used with each load balancer, we performed runs with 1, 2, 4, 8, 16, and 32 processors. The

speedup over the no load balancer situation are plotted in Figure 5 for the `Paraffins`(28) and `Queens`(12) programs. The gain of performance of the Minima over the no balancer situation *Nop* indicates the benefit of a simple initial distribution of work for that program.

5.4.1 Overheads for Supporting a Multi-threaded Environment

The *uni-node support efficiency* or USE factor [22] is the ratio of sequential execution time and the elapsed time for one-node parallel execution. An ideal 100% use-factor indicates minimum overheads imposed by the multi-threaded environment. It also indicates that the program has enough parallelism to hide the latencies of the multi-threaded operations. A unity USE factor also indicates good absolute speedup, and the opportunity for efficient load balancing.

Table 5 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`. For `Queens` both the absolute and the relative speedups with the *Rand* balancer are higher than their counterparts for the *His* balancer. As a consequence *Rand*'s speedup is better for `Queens`. However, for `Tomcatv` *Rand* delivers lower performance than *His*.

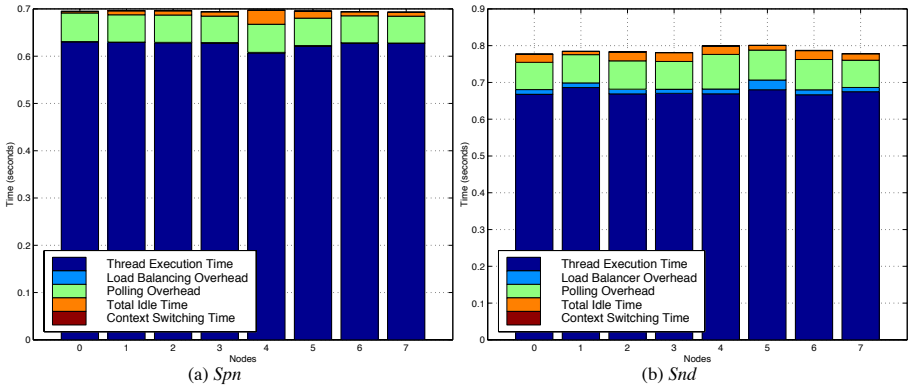
Benchmark	Balancer	Absolute Speedup	Relative Speedup	USE-factor %
`Fibonacci`(28)	His	0.88	18.49	4.77
	Rand	1.03	17.47	5.89
`Queens`(12)	His	25.95	28.74	90.29
	Rand	27.47	30.91	88.89
`TSP`(10)	His	28.66	31.31	91.55
	Rand	29.21	31.89	91.62
`Matrix`(1024X1024)	His	24.71	26.23	94.18
	Rand	17.29	16.77	103.07
`Tomcatv`(257)	His	8.69	13.88	62.63
	Rand	0.84	1.37	61.57
`Paraffins`(28)	His	8.73	31.48	27.72
	Rand	8.84	31.88	27.74

Table 5: Absolute and Relative speedups for a **32 node** run. The USE factor is the *Uni-node Support Efficiency*, and is the ratio of absolute speedup to relative speedup.

5.4.2 Distribution of Total Elapsed Time

Figures 6 and 7 display the break-up of the total elapsed time in each processing node for the `Queens`(12) program on a run using eight processors. [2] In this experiment less processors are used, thus the elapsed times are longer than reported in earlier experiments. The graph in Figure 6 shows the time break-up for the *Spn* and *Snd* balancers. For `Queens`, the work stealing *Spn* balancer works best because of the recursive nature of this program. Quite early in the execution every node has enough work to keep itself busy, and thus no more load balancer related traffic is generated. In balancers with a sender component, each

[2] The order of the colors in the legends of these graphs is the reverse of the colors that appear in the bars. In all graphs the largest portion of the time is spent on thread execution, and the second largest is polling overhead. The small amounts of time that appear at the very top of each bar are the portions of idle time in each node.

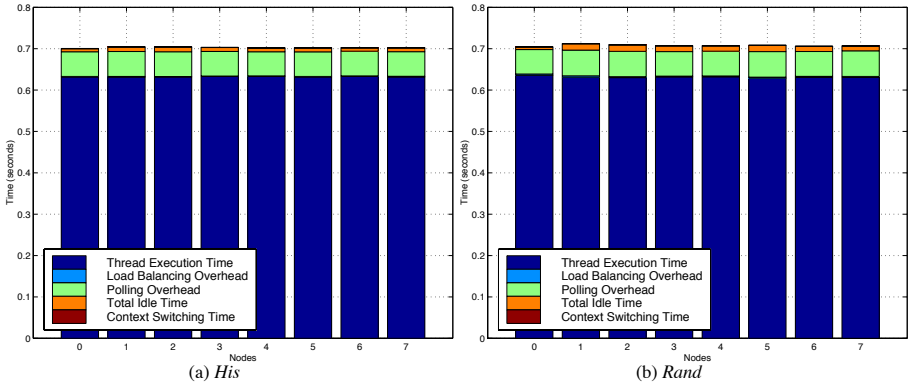


(a) *Spn* (b) *Snd*

Fig. 6: A Distribution of Elapsed Time for `Queens`(12) on 8 nodes for the *Spn* and *Snd* balancers.

node that is busy will attempt to do load balancing by either sending tokens, or by sending load probes (as is the case for the *Rand*) to other nodes. For instance, the *Snd* balancer sends tokens to remote nodes. When all the nodes in the system are rich, these futile attempts to transfer load waste CPU and network resources. One possible improvement to the *Rand* balancer is to detect the situation in which all nodes are rich and disable its sender component.

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. For a breakup of the total execution time for the other benchmarks, please refer to [11].



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

Fig. 7: A Distribution of Elapsed Time for `Queens`(12) on 8 nodes for the *His* and *Rand* balancers.

6 Related Work

Many existing multithreading systems [11,12] are software emulations based on off-the-shelf hardware and compiler technologies. Software multithreaded systems can be broadly classified into language and library-based systems. Examples of language-based multithreaded systems include EARTH [10], Cilk [7], TAM [6], Concert [14], Java [8] and C+- [4]. Each of these systems, except for Java, supports non-blocking, non-preemptive threads. An exception in this category of multithreaded systems is the Java programming language.

Examples of library-based systems include Nano-threads [2], Ariadne [16], Opus [17], Structure Thread Library [24], and Active Threads [28].

Cilk is an algorithmic multi-threaded language currently designed for symmetric multiprocessors (SMP's) [3]. Central to Cilk's development is the scheduling of multi-threaded computations using a work-stealing mechanism [3]. The Cilk compiler and runtime system jointly play an active role in dynamic load balancing[3]. The Cilk runtime system [3] 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 [10], and also reported in the ADAM architecture [15]. The generation of two clones for every Cilk procedure is an application of the *work-first* principle [7].

The Cilk threading model is very amenable for the solution of divide-and-conquer problems, and is most suited for fully-strict computations [3]. 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 Threaded Abstract Machine project [6] 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 to exploit considerable inter-thread locality. TAM was one of the first multi-threaded systems that were built through software emulation with minimal hardware support. Load distribution of workload onto processors is managed by the TAM compiler [20], whereas in EARTH the workload is dynamically distributed at runtime by the load balancer.

7 Conclusion

We demonstrated that a fine grain multi-threaded system can be implemented with an standard off-the-shelf processor, memory system and compiler technology of a distributed memory architecture. Through a combination of clever pre-compiler transformations and a well-crafted runtime system, the EARTH system delivers non-preemptive, fine-grain multi-threading at low cost. Our detailed experimental study of various load balancing policies indicates that a randomized policy delivers good performance on a 32 node parallel system.

Presently the work described in this article is being ported onto the EARTH-Beowulf system, a Linux platform. Further study of the runtime system performance, particularly the balancers in a shared memory environment (SMP) is in progress. A logical next step in designing better load balancers for the EARTH system is the design of adaptive load balancers; these balancers change balancer policy dynamically at runtime, based on application workload, and node availability.

The results presented here help understand the influence of load balancing policies in the performance of multi-threaded systems. These results should also be applicable to similar parallel systems based on multi-threading and will hopefully allow future systems to achieve better performance for a broad range of applications.

[3] This is unlike EARTH, where dynamic load balancing is a purely runtime system activity.

8 Acknowledgements

The authors are greatful to members of the ACAPS Lab, McGill University, and the CAPSL Lab, University of Delaware for their insight, ideas and help. The authors thank the Cornell Theory Center, the access to their IBM SP-2 system. We 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. The authors also acknowledge the partial support from DARPA, NSA, NSF (under grants NSF-CISE-97263 88 and NSF-MIPS-9707125) and JPL-NASA. The initial EARTH work was partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

1. 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.
2. 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*, July 1997.
3. 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, August 1996.
4. 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.
5. 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*, January 1999.
6. 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*, April 1991.
7. 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*, June 1998.
8. James Gosling and Henry McGilton. The Java Language Environment. In *A White Paper, Sun Microsystems, California, USA*, pages 1–95, May 1996.
9. 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 *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 12–23, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.
10. 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. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.
11. Prasad Kakulavarapu. Dynamic Load Balancing Issues in the EARTH Runtime System. Technical report, School of Computer Science, McGill University, Montreal, Québec, 1999. Master's thesis, 1999.
12. Prasad Kakulavarapu and José Nelson Amaral. A survey of load balancers in modern multithreading systems. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 10–16, Natal, Brazil, September 29–October 2, 1999.

Brazilian Computer Society.

13. 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 *In Proc. of the 19th IEEE International Performance, Computing, and Communications Conference-IPCCC 2000, February 20-22, 2000 , Phoenix, Arizona, USA (also available as Technical Memo 35, CAPS Lab, University of Delaware)*, November 1999.
14. 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, August 1996.
15. 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.
16. 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, March 1996.
17. Piyush Mehrotra and Matthew Haines. An Overview of the Opus Language and Runtime System. Technical report, May 1994.
18. Michael David Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. In *Ph. D Thesis, University of California, Berkeley, California*, 1996.
19. 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.
20. 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.
21. Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. In *IEEE Computer*, pages 33 – 44, December 1992.
22. Kevin B. Theobald. EARTH - an Efficient Architecture for Running THreads. Technical report, School of Computer Science, McGill University, Montreal, Québec, 1999. PhD thesis, 1999.
23. 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*, March 1998.
24. 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*, August 1998.
25. 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.
26. Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 54–58, Orlando, Florida, January 9–13, 1999. IEEE Computer Society.
27. 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.
28. Boris Weissman. Active Threads: an Extensible and Portable Light-Weight Thread System. Technical report, September 1997.