A Multi-Threaded Runtime System for a Multi-Processor/Multi-Node Cluster

Christopher Jason Morrone (morrone@capsl.udel.edu)
Computer Architecture and Parallel Systems Laboratory (CAPSL)
Dept. of Electrical and Computer Engineering, Univ. of Delaware
Newark, DE, USA

José Nelson Amaral (amaral@cs.ualberta.ca)
Department of Computing Science, University of Alberta
Edmonton, AB, Canada

Guy Tremblay (tremblay.guy@uqam.ca)
Dépt. d'informatique, Université du Québec à Montréal
Montréal, QC, Canada

Guang R. Gao (ggao@capsl.udel.edu)
CAPSL, Univ. of Delaware, Newark, DE, USA

Abstract.
We designed and implemented an EARTH (Efficient Architecture for Running THreads) runtime system for a multi-processor/multi-node, cluster. For portability, we built this runtime system on top of Pthreads under Linux. This implementation enables the overlapping of communication and computation on a cluster of Symmetric Multi-Processors (SMP), and lets the interruptions generated by the arrival of new data drive the system, rather than relying on network polling. We describe how our implementation of a multi-threading model on a multi-processor/multi-node system arranges the execution and the synchronization activities to make the best use of the resources available, and how the interaction between the local processing and the network activities are organized.

Keywords: multi-threading, cluster computing

1. Introduction

This paper describes the design and implementation of a runtime system for the multi-threaded EARTH architecture. This is the first completely functional implementation of the EARTH system on a cluster of symmetric multi-processor (SMP) nodes. Our runtime system is designed for easy portability across Beowulf systems constructed with various processor nodes. It uses standard Unix sockets for inter-node communication and splits the tasks performed in the EARTH system — thread execution, communication, and synchronization — into three separate modules: an execution module, a sender module, and a receiver module. As discussed in Section 4, this organization of the runtime sys-

tem is fundamental for an efficient, portable, and deadlock-free runtime system.

This paper is organized as follows. Section 2 describes the EARTH programming model and its programming language, Threaded-C (release 2.0), a revised version of the language used to write the programs run on the EARTH system. Section 3 briefly describes the EARTH architecture model and the role of the runtime system (RTS). Section 4 discusses the new design of the RTS. Performance results (speedup curves) are then presented in Section 5, comparing our implementation of the EARTH runtime system for SMP clusters with an earlier implementation of such system. Results are presented for two clusters, one with 16 single processor nodes, and another with 64 dual-processor nodes. Finally, Section 6 briefly presents related work.

2. The EARTH Programming Model and its Programming Language

This section presents the EARTH programming model and how it is embodied in a specific programming language, called Threaded-C. EARTH's programming model has its origin in the dataflow model of computation. In a pure dataflow model fine-grain parallelism is supported by representing programs as graphs where each node is associated with a single instruction and arcs indicate data exchanged between instructions. EARTH's programming model also rests on the notion of a dataflow graph, except that nodes can be associated with sequences of instructions whereas arcs are simply synchronization signals indicating dependencies.

An important characteristic of EARTH's programming model is its two-level hierarchy formed by threaded functions and fibers. Threaded functions are instantiated by the parallel activation of procedures. Threaded functions are thus similar to the threads found in Java [12] or POSIX [5]. A distinguishing characteristic of EARTH's threaded functions, however, is that they can themselves contain an additional level of parallelism, called *fibers*. A fiber is an independent and *lightweight* thread of execution that corresponds strictly to a segment of code *inside a threaded function*. Furthermore, since the different fibers of a threaded function share the same context, viz., the activation frame of their parent procedure, they allow for rapid context switch and, thus, for fine-grain parallelism.

Fibers possess the following characteristics:

- Fibers are scheduled using a dataflow approach: a fiber becomes ready to execute when it has received all appropriate signals.

```
 1  THREADED fib( int n, int *GLOBAL result, SPTR done )
 2  {
 3    int r1, r2;
 4
 5    if (n <= 1) {
 6      PUT_SYNC( 1, result, done );
 7      TERMINATE;
 8    } else {
 9      TOKEN( fib, n-1, TO_GLOBAL(&r1), TO_SPTR(READY) );
10      TOKEN( fib, n-2, TO_GLOBAL(&r2), TO_SPTR(READY) );
11    }
12
13    FIBER READY <* 2 *> {
14      PUT_SYNC( r1+r2, result, done );
15      TERMINATE;
16    }
17  }
```

Figure 1. Threaded-C recursive procedure for computing the nth Fibonacci number

- Instructions within fibers execute sequentially based on the underlying language semantics (in our case, C).
- Fibers execute in a non-preemptive manner. In other words, a fiber is never interrupted and must never block.

The Threaded-C language was designed to support the two-level threading hierarchy of EARTH as well as the appropriate fiber semantics (dataflow scheduling and non-preemptiveness). Threaded-C evolved from an initial version, where some low-level details had to be explicitly specified [18] (e.g., declaration of synchronization slots, numeric values for identifying fibers and slots), to a more recent version [19] designed to simplify the language and make it easier to use. It is important to stress, however, that both versions of the language allow a programmer to have complete control over the decomposition into threads and fibers and on communications and synchronizations.

Figure 1 presents a recursive procedure `fib`, written in Threaded-C (release 2.0), for computing the nth Fibonacci number. `THREADED` (Line 1) means that parallel activations of this procedure can be created using `TOKEN` (Line 9 and 10). Alternately we could have used `INVOKE` to create tokens.

One key characteristic of the EARTH model, apparent also in Threaded-C, is its underlying memory model. Although EARTH supports a global address space with uniform addressing, it does not presume that remote locations can be accessed using ordinary instructions. In

Threaded-C, the notion of `GLOBAL` handle (pointer to a possibly remote location) is thus introduced and special instructions are used to transfer data to/from remote locations.

Based on these notions, the example can be explained as follows:

- The initialization fiber (Lines 5–11), executed as soon as `fib` starts, first checks whether recursive calls must be performed or not.
- In the base case (Lines 6–7), the result is returned immediately using a `PUT_SYNC` instruction. When the transfer is complete, a signal is sent to the synchronization slot `done`. The thread then terminates (Line 7), so the activation frame can be deallocated.
- In the recursive case (Lines 8–11), procedure calls are done using the `TOKEN` instruction, which means the RTS will select the node(s) where the procedures will be executed. These invocations will return their result using distinct local variables (`r1` and `r2`) but will both send a completion signal to the same slot (`TO_SPTR(READY)`).
- A synchronization slot is implicitly associated with a fiber of the same name (Line 13). The "`<* k *>`" construct indicates how many signals must be received before a fiber becomes enabled (ready for execution). The result is also returned using a `PUT_SYNC`.

Threaded-C (release 2.0) provides support for multi-threaded programming of SMP machines. On such machines, there can be multiple fibers, all executing at the same time, accessing the same local memory. Mutual exclusion mechanisms must thus be provided:

1. Mutually exclusive fibers: A fiber declared as `EXCLUSIVE` will always be the *unique exclusive fiber* among a given thread to be executing at any given time (similar to Java's `synchronized` [12]).
2. Atomic mailboxes: This data type provides a form of non-deterministic merge operator, as typically found in dataflow models, that is, a location where multiple messages from different sources can be stored until retrieved by consumers. The key operations are the followings (see [19] for additional operations):
   - `INIT_MAILBOX(MAILBOX *mb, SLOT s);`: Allocates a mailbox on the local processor and associates with it a synchronization slot `s` where a signal will be sent each time a new item arrives.
   - `DROP_IN(MAILBOX *GLOBAL mb, void* item, int nb_bytes);`: Transfers an item (of size `nb_bytes`) to the (possibly) remote mailbox `mb`. When the item has been put into the destination mailbox, a signal is sent to its associated sync slot.

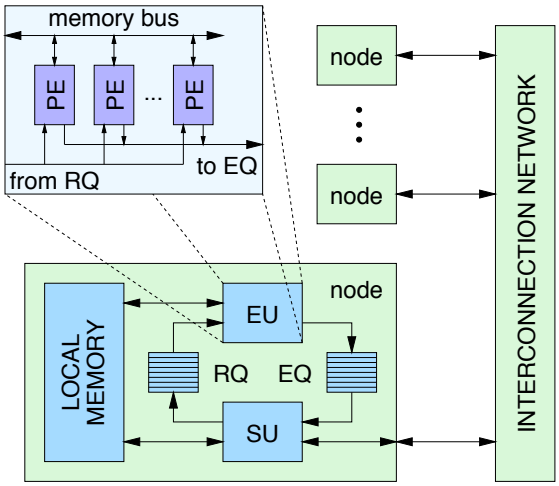

Figure 2. EARTH Architecture

— `RETRIEVE_ITEM(MAILBOX mb, void* item);`: Retrieves an (arbitrary) item from `mb`. Contrary to `DROP_IN`, this operation must be executed on the node where the mailbox has been allocated.

Threaded-C dataflow style communication and synchronization operations can be and have been used to define other synchronization mechanisms, for example, locks, semaphores, I-structures, communication channels, parallel reduction boxes, etc.

3. The EARTH Architecture Model and the Role of the Runtime System

Figure 2 shows the general structure of the EARTH architecture model. In this model, processing nodes are interconnected via a network and each processing node contains a synchronization unit (SU) and one or more execution units (EU). The EU is responsible for doing the "*useful work*", that is, for executing fibers. The SU is in charge of synchronization, inter-node communication, scheduling, and load balancing. All of the local EUs and the SU communicate with each other through the ready queue (RQ) and the token queue (TQ).

Because fibers correspond to sequences of instructions and are non-preemptive, and because the SU takes care of all synchronization tasks, instructions from fibers can be executed using a regular instruction pipeline. An EARTH machine can thus be built using off-the-shelf processors [10]. For instance, the experimental results presented in Section 5 were obtained on two different Beowulf machines: Ecgth-

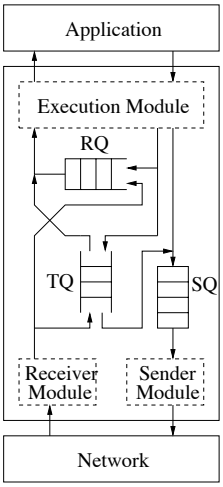

Figure 3. EARTH Runtime System

eow, a cluster at Michigan Technological University consisting of 64 dual-processor nodes, and Earthquake, a cluster at the University of Delaware consisting of 16 single-processor nodes. Other implementations have also been built for various machines: MANNA, network of Sun workstations, IBM SP-2, etc.

When using standard processors to build an EARTH machine, the SU and the various elements comprising an EARTH node must be implemented in software. Handling these various tasks, including creating threads, scheduling fibers, and handling network communication, is the task of the runtime system (RTS). In the next section, we present our new design for this RTS.

4. The Design of the New Runtime System

Our goal is to design an EARTH runtime system that is portable, makes efficient use of existing standard network interfaces, uses all the processing resources available in SMP Beowulf clusters, and delivers good performance.

The general structure of our new EARTH runtime system is shown in Figure 3. The Execution Module executes fibers and also takes on the responsibilities of intra-node scheduling, synchronization, communication and load balancing, tasks which were performed strictly by the SU in previous implementations of EARTH. The Receiver (resp. Sender) Module handles incoming (resp. outgoing) messages. The Token Queue

(TQ) contains work that may either be performed by a processor within the local node, or that might be sent to a different node for execution. The Ready Queue (RQ) contains fibers that must be executed locally, and the Sender Queue (SQ) contains the outgoing messages.

In the following paragraphs, we describe the interface between the RTS and the network, and how our design for the RTS benefits from the resources available in an SMP machine. We also discuss the tradeoff between polling and interrupts, blocking I/O, potential deadlocks in a RTS, and the use of multiple processors in each node.

4.1. The Case for Standard Unix Sockets

We chose the convenience of end-point communication provided by Unix sockets to establish the communication channels between multiple SMP processor nodes. Sockets provide an easy to use Application Programming Interface (API) that is consistent across many operating systems and networking hardware.

Earlier implementations of EARTH used custom network interfaces that allowed more efficient use of the network but were difficult to port to newer hardware and operating systems [17, 11].

4.2. To Poll or Not To Poll

We know three options to implement the communication between the runtime system and the network interface: interrupts, polling, and polling-watchdog. Interrupts are usually not desirable in a multi-threading system because they interrupt the running thread and lead to a context switch. The preferred method is for the runtime system to poll the network between the execution of threads, and thus avoid unnecessary context switching. A third method, *polling-watchdog*, developed especially for EARTH [14], mixes polling and interrupts in the following way: the runtime system polls the network between thread context switching, but when a message arrives, a timer starts counting; if the message is not handled within a given amount of time, the network interface interrupts the runtime system. The advantage of polling-watchdog is that it prevents a thread containing a long running loop from making the node where it is running oblivious to what is happening in the remaining nodes of the cluster. To implement a polling-watchdog, however, it must be possible to program the network interface to implement the appropriate time-out mechanism. This was done in earlier versions of the EARTH runtime system, but made those systems less portable.

Since Unix sockets are used for our inter processing node communication, a polling approach would use the *select()* system call, making

the kernel perform a linear search on its socket structures to identify which socket has an incoming message. In a large cluster with many open sockets, polling can thus become expensive (between 2,500–15,000 processing cycles).

The drawback of interrupts is that they happen asynchronously with the thread context switching in a multi-threading system. Nevertheless, our decision to use Unix sockets makes interrupts unavoidable and care must be taken when handling them. When a message arrives, the Ethernet card raises a hardware interrupt that the CPU handles by stopping the currently running process. The OS interrupt handler decodes the interrupt and runs the appropriate hardware driver. When the driver is finished, the running process is allowed to continue at the point where it was interrupted. In our design, the kernel informs the runtime system of the arrival of a message and the runtime system immediately takes the actions required to process the message within the EARTH model before allowing the interrupted thread to resume execution.

4.3. Blocking vs. Non-Blocking I/O

After the runtime system is notified that a message arrived, it needs to transfer the message's content, using a *read()* operation, from the socket buffer in kernel space into a buffer in user space. When the runtime system needs to send a message, it issues a *write()* operation to a socket. Both the read and the write operation behavior are affected by the use of blocking or non-blocking I/O. Such effect appears when a read requests more bytes than currently available in the socket buffer, or when the buffer overflows in the case of a write operation. In a non-blocking I/O system, any read or write operation will return immediately with the number of bytes that were successfully read/written. On the other hand, in a blocking I/O system, a call will not return (i.e., will block) until all bytes have been read/written.

Modern systems use default socket buffer sizes of 8192–61440 bytes [16]. Because modern processors are much faster than any available networking technology, these small buffers often become full. Thus the potential blocking situation can be frequent. If blocking I/O is used in a system where the same OS process handles both the network activities and the execution of EARTH threads, precious CPU cycles would be wasted when an I/O operation blocks. In our current design we opted for a blocking I/O system but created two separate modules (pthreads) for the network activities. Thus, a blocked I/O operation does not block the processor indefinitely.

4.4. A Deadlock-Free Runtime System

When blocking I/O is used, a potential deadlock condition may arise when both the socket send buffer on one end of a link and the socket receive buffer on the other end of the same link become full. Potentially both nodes might be blocked in a *write()* operation to the send buffer and neither will read from the receive buffer to allow communication to proceed. More general deadlock situations can be described considering multi-threaded programs that have cycles of inter-node dependencies.

Our solution, as shown in Figure 3, is to create two separate modules in each processing node: a sender module and a receiver module. Now, when the sender module blocks, the processor can switch to the receiver module and read any incoming data that is there. If the scheduling is fair with respect to the receiver module, this will not lead to a deadlock situation, even if the execution module is allowed to proceed when the sender module blocks.

4.5. Benefiting from SMP

The RTS described here uses the blocking mode of access to sockets. Polling is avoided by the use of blocking calls to *select()*, which will only return when incoming traffic has arrived in a socket's receive buffer.

To avoid wasting cycles when blocking occurs — waiting for incoming network reads or waiting for outgoing network writes — the parts of the RTS that might block are decomposed into separate modules (pthreads). Furthermore, these threads are distinct from the thread responsible for running fibers.

The part of the RTS which runs fibers is called the *Execution Module* (EM). The two modules which, together, handle all networking activity, are known as the *Sender Module* (SM) and the *Receiver Module* (RM). These three modules are implemented as separate threads using POSIX threads (pthreads). The networking modules are only active when there is network traffic that needs to be handled; at all other times, they avoid wasting CPU cycles by going to sleep.

With the new modular design of the RTS, it becomes easier to provide SMP node support. Not only can the EM, SM, and RM run concurrently, but also we might implement multiple EMs in the same processing node, with concurrent executions in multiple processors.

When multiple EMs are active, each has its respective Ready Queue, although they all share a Token Queue and a Send Queue. All of the modules are implemented as pthreads, and therefore share the same memory space. Intra-node communication is accomplished simply and efficiently through memory reads and writes.

5. Experimental Results

5.1. The Experimental Platform

In order to evaluate the performance of our runtime system, we installed it on two Beowulf clusters: "Earthquake" operated by the Computer Architecture and Parallel Systems Laboratory at the University of Delaware, and "Ecgtheow" operated by the Computational Science and Engineering program at Michigan Technological University and sponsored by the NASA HPCC/ESS project. Earthquake is sixteen 500MHz Pentium III processor nodes with 128MB of RAM. Ecgtheow has 64 nodes, each with dual Pentium Pro processors (a total of 128 processors) and 128MB of RAM. The interconnection network for both clusters is Fast Ethernet. For our RTS implementation, the most important distinction between these two clusters is the single processor nodes in Earthquake and the dual processor nodes in Ecgtheow.

We also ran two versions of the runtime system to evaluate the influence of the runtime system design and implementation on performance. RTS 1.2 uses a polling method to access the network, implements non-blocking sockets and concentrates all the activities (thread execution, sender, and receiver) in a single module. By contrast, RTS 2.0 uses interrupts to access the network, implements blocking sockets, and separates the execution of threads, the sending, and the receiving activities of the network into three separate threads.

5.2. Test Programs

We used three programs to evaluate our implementation of the runtime system: (1) a recursive implementation of Fibonnacci in which each non-base call to the Fibonnacci function generates two distinct recursive calls (see Figure 1 (p. 3) for a Threaded-C version of this procedure); (2) a recursive, non-throttled implementation of N-queens; and (3) ATGC (Another Tool for Genome Comparison), a multi-threaded implementation of a dynamic programming algorithm for sequence comparison [15].

Figure 4 presents the speedup curves for runs of `fib(32)` (a recursion with 4.3 billion leaves) on Ecgtheow and Earthquake with both RTS 1.2 and RTS 2.0. The recursive Fibonnaci implementation is not throttled because it is used to test the runtime system ability to handle applications that generate a large number of threads. Observe that, for all speedup curves presented in this section, Ecgtheow has two processors in each processing node while Earthquake has only a single processor in each node. RTS 2.0 (1 EM) is a version of the runtime system that

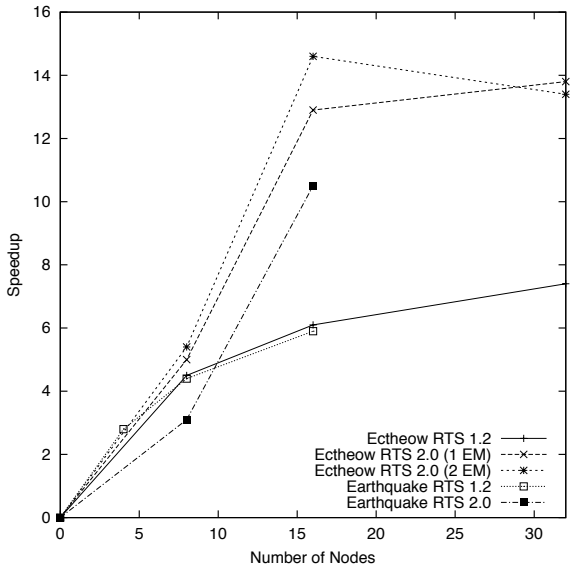

Figure 4. Speedup curves for Fibonnaci

implements a single execution module in each processing node, while RTS 2.0 (2 EM) implements two execution module per processing node.

For clusters with more than 10 processing nodes, RTS 2.0 outperforms RTS 1.2 in both machines. When two EMs are used in Ecgtheow, RTS 2.0 delivers a speedup of 15 in 16 dual-processor nodes, while RTS 1.2 has a speedup of only 10.5. Observe that on Earthquake when only eight single-processor nodes are used RTS 2.0 underperforms RTS 1.2. This is because, with a single processor per node, the cost of switching between the multiple modules of RTS 2.0 becomes significant. This cost, however, is amortized by the more efficient network interface when all 16 nodes of Earthquake are used.

The goal of the N-queens program, a recursive backtracking algorithm that is representative of some typical highly parallel applications, is to determine the number of ways in which n queens may be placed on an $n \times n$ chessboard so that no queen is in a position to attack another. Figure 5 shows the speedup curves for N-queens on a 12×12 board. RTS 2.0 shows clear improvement over RTS 1.2. Also important to note is the decline in speedup that happens with RTS 1.2 when the program runs on more than 8 nodes. Increase in speedup clearly tapers off at 8 nodes for RTS 2.0 as well, but speedup continues to increase gradually as nodes are added, and there is no adverse effect on runtime. [1]

[1] Much better speedup curves for N-queens(12) can be obtained when the program is throttled at an adequate level and stops generating new threads. In this

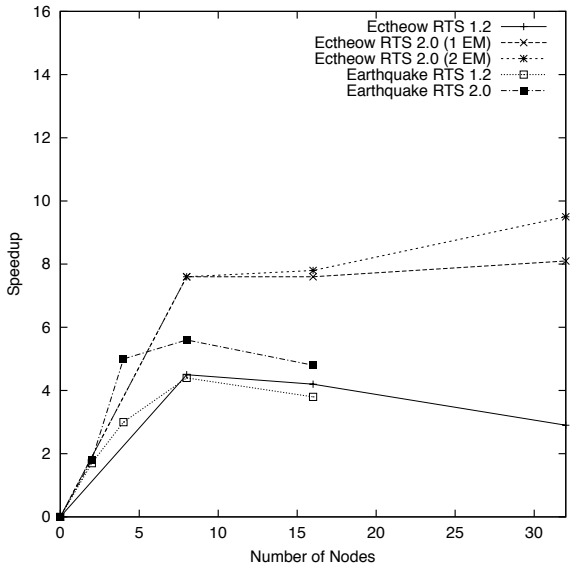

Figure 5. Speedup Curves for N-Queens(12)

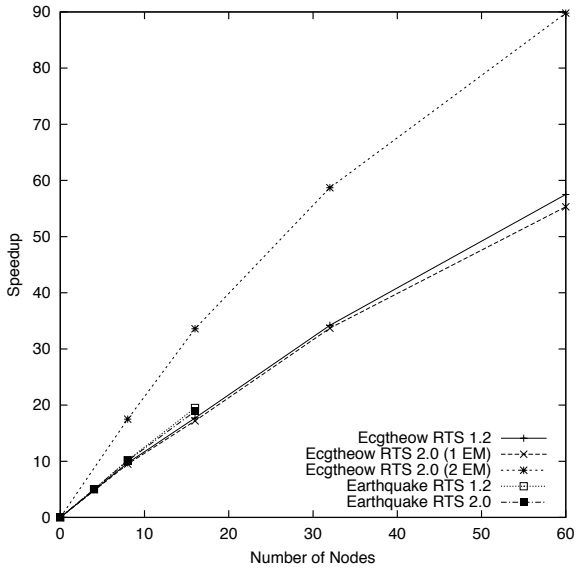

Figure 6. Speedup curves for ATGC

The third benchmark — ATGC — is a dynamic programming algorithm for DNA sequence comparison. In this program run, two random DNA sequences, both of size 40K base pairs, were compared. In the graph of Figure 6, the speedup curves for RTS 1.2 and RTS 2.0 using one EM are very similar. ATGC is an application which is far more CPU intensive than network intensive. Likewise, when ATGC is run on Earthquake, the speedup curves for RTS 1.2 and RTS 2.0 are very similar. The advantage of the new design in RTS 2.0 is made more evident when a second EM is activated. RTS 2.0 is able to fully utilize both processors in every node of Ecgtheow, which results in a speedup curve nearly double that of the other two curves. [2]

6. Related Work

The EARTH model has its origin in the argument-fetching dataflow model, a dataflow model *without flow of data* [9]. EARTH also has been influenced by early work in multi-threading parallel architectures [13]. Earlier implementations of the EARTH system are described in [10, 17, 11]. The ATGC program is described in [6, 15]. Recent projects most relevant to our research are Cilk and MPI.

Like EARTH, Cilk is a C-based multithreaded language and runtime system [2]. However, in its initial design, Cilk targetted exclusively shared memory machines. Cilk uses a provably good "work-stealing" scheduling algorithm and follows a "work-first" principle. Cilk concentrates on minimizing overheads that contribute to work, even at the expense of overheads that contribute to the critical path [8]. Cilk-NOW is an implementation of Cilk for networks of workstations [1, 3]. It transparently manages resources, provides transparent fault tolerance, and implements "adaptive parallelism" which allows a Cilk application to run on a set of workstations that may grow and shrink throughout program execution.

Cilk's underlying programming model is limited to divide-and-conquer parallelism and does not support the two level hierarchy of threaded functions vs. fibers that makes Threaded-C a multithreaded language that can express parallelism at varying level of granularity, including irregular fine-grain parallelism.

MPI is a standard interface for the message passing paradigm that seeks to combine the most attractive features of existing message pass-

study we do not throttle because our goal is to assess how well the RTS handles high volumes of tokens.

[2] Notice that when executing in 60 nodes Egtheow is using 120 processors. Thus the speedup o f 90 in Figure 6 is sublinear.

ing systems [7]. MPI is a widely accepted industry standard that makes it possible to write portable parallel programs. MPI's programming model, contrary to Threaded-C, supports only coarse-grain parallelism with all processes having to be created statically. On the other hand, MPI provides a rich set of operations for global communication, e.g., broadcast, scatter, and gather [4].

7. Conclusion

This paper has presented the new design of the runtime system for the EARTH multithreaded architecture. The intended target machines for this new RTS are modern multi-nodes systems with multiple processors per node (SMP clusters). The RTS has been designed with the goal of being portable, yet being able to benefit efficiently from the power of multiple processors per node. In order to do this, the RTS implementation uses multiple threads of execution.

Acknowledgments

The authors would like to thank current and former members of CAPSL at the University of Delaware for valuable ideas exchange. Special thanks to Kevin Theobald for the N-queens code, and to Juan del Cuvillo and Wellington Martins for the ATCG code. The authors also acknowledge the partial support from DARPA, NSA, NSF (under grants NSF-INT-9815742 and NSF-CSA-0073527), and NASA. The initial EARTH work was partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

1. R. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, MIT, Dept. of EE and CS, Sept. 1995.
2. R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.
3. R.D. Blumofe and P.A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Symposium*, California, 1997.
4. J. Bruck and al. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. In *7th Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 64–73.

5. D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
6. J.B. del Cuvillo, W.S. Martins, G.R. Gao, W. Cui, and S Kim. ATGC: Another tool for genome comparison. In *International Conference on Computational Molecular Biology - RECOMB*, April 2001.
7. Message Passing Interface Forum. MPI: A message-passing interface standard (version 1.0). Technical report, May 1994. URL http://www.mcs.anl.gov/mpi/mpi-report.ps.
8. M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montréal, Québec, June 17–19, 1998. *SIGPLAN Notices,* 33(6), June 1998.
9. G.R. Gao and R. Yates. The argument-fetching dataflow architecture project: A status report. In *Can. Conf. on Elec. and Comp. Eng.*, Montreal, Sept. 1989.
10. H.H.J. Hum, K.B. Theobald, and G.R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proc. of the 8th IEEE Intl. Parallel Processing Symp. (IPPS '94), Cancùn, Mexico*, pages 288–294, April 1994.
11. P. Kakulavarapu, O. Maquelin, and G.R. Gao. Design of the runtime system for the Portable Threaded-C language. CAPSL Technical Memo 24, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, July 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.
12. D. Lea. *Concurrent Programming in Java — Design Principles and Patterns (Second Edition)*. Addison-Wesley, 2000.
13. B. Lee and A.R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, 27(8):27–39, 1994.
14. O. Maquelin, G.R. Gao, H.H.J. Hum, K.B. Theobald, and X.-M. Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *23rd Annual International Symposium on Computer Architecture*, pages 178–188.
15. W.S. Martins, J.B. del Cuvillo, F.J. Useche, K.B. Theobald, and G.R. Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *Pacific Symposium on Biocomputing*, Jan. 2001.
16. W.R. Stevens. *UNIX Network Programming, Networking APIs: Sockets and XTI*, volume 1. Prentice-Hall, Upper Saddle River, NJ, 1998.
17. K.B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, Montréal, Québec, May 1999.
18. K.B. Theobald, J.N. Amaral, G. Heber, O. Maquelin, X. Tang, and G.R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, University of Delaware, March 1998.
19. G. Tremblay, K.B. Theobald, C.J. Morrone, M.D. Butala, J.N. Amaral, and G.R. Gao. Threaded-C language reference manual (release 2.0). CAPSL Technical Memo 39, University of Delaware, Sept. 2000.