Implementation of the EARTH programming model on SMP clusters: a multi-threaded language and runtime system

G. Tremblay[1], C.J. Morrone[2], J.N. Amaral[3], G.R. Gao[1]

[1]*Dépt. d'informatique, Univ. du Québec à Montréal, Montréal, QC, Canada*
[2]*Computer Architecture and Parallel Systems Laboratory (CAPSL), Dept. of Electrical and Computer Engineering, Univ. of Delaware, Newark, DE, USA*
[3]*Dept. of Computing Science, Univ. of Alberta, Edmonton, AB, Canada*

SUMMARY

This paper describes the design and implementation of an EARTH runtime system for a multi-processor/multi-node cluster. The Efficient Architecture for Running THreads (EARTH) model was designed to support the efficient execution of parallel (multi-threaded) programs with irregular fine-grain parallelism using off-the-shelf computers. Implementing an EARTH runtime system requires an explicitly threaded runtime system. For portability, we built this runtime system on top of Pthreads under Linux and used sockets for inter-node communication. Moreover, in order to make the best use of the resources available on a cluster of Symmetric Multi-Processors (SMP), this implementation enables the overlapping of communication and computation.

We used Threaded-C, a language designed to implement the programming model supported by the EARTH architecture. This language allows the expression of various levels of parallelism and provides the primitives needed to manage the required communication and synchronization. The Threaded-C programming language supports irregular fine-grain parallelism through a two-level hierarchy of threads and fibers. It also provides various synchronization and communication constructs that reflect the nature of EARTH's fibers — non-preemptive execution with data-driven scheduling — as well as the extensive use of split-phase transactions on EARTH to execute long latency operations.

KEY WORDS: multi-threading, cluster computing, parallel programming language

*Correspondence to: José Nelson Amaral, Department of Computing Science, University of Alberta, Edmonton, AB, T6G 2E8, Canada

1. Introduction

This paper describes the design and implementation of the EARTH programming model on a cluster formed by symmetric multi-processor (SMP) nodes. The EARTH model is designed to support the efficient execution of parallel (multi-threaded) programs with irregular fine-grain parallelism using off-the-shelf computers [?, ?]. In order to allow the expression of various levels of parallelism and to make communications and synchronizations explicit, the Threaded-C language was designed to program this architecture. This language evolved along with earlier developments of the EARTH model [?, ?, ?]. In this paper we discuss the new language features that we introduced in a revised version of the Threaded-C language (release 2.0) in order to make the language easier to use and to allow the correct functioning of Threaded-C programs running on SMP clusters.

Because of EARTH's special characteristics (described in Section 2), the implementation of the EARTH model requires an explicitly threaded runtime system. Earlier versions of the runtime system were developed for distributed memory computing platforms [?]. In this paper we describe the first completely functional implementation of the EARTH system on a cluster of symmetric multi-processor (SMP) nodes. This runtime system is designed for easy portability across 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 system is fundamental for an efficient and portable runtime system. This organization is also important to avoid deadlocks when using blocking I/O for inter-processor communication in the runtime system implementation.

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

2. The EARTH Programming Model and its Programming Language

In this section, we present the EARTH programming model and discuss how it is expressed in the Threaded-C programming language. EARTH's programming model has its origins 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 is also based on the notion of a dataflow graph, except that the graph nodes are associated with *program segments* (called *fibers*) whereas arcs are simply synchronization signals (without an

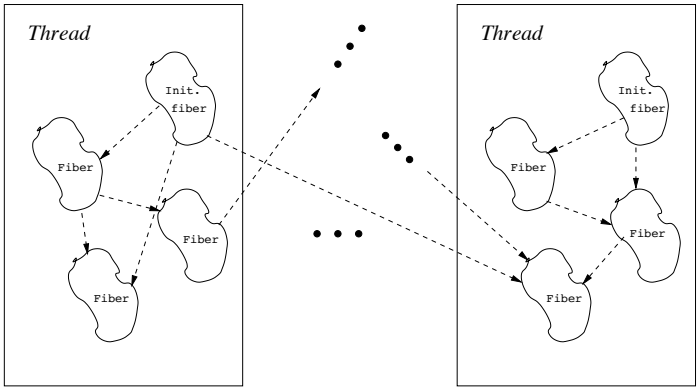


Figure 1. An EARTH program with threads and fibers

associated flow of data) indicating dependencies between fibers. Furthermore, EARTH's model supports two levels of parallelism.

EARTH's two-level hierarchy of threads and fibers

An important and distinguishing characteristic of EARTH's programming model is its two-level hierarchy of parallelism obtained using *threaded functions* and *fibers*, as illustrated in Figure 1. Threaded functions are instanced by the parallel activation of C functions — note that, in the remaining discussion, the common C terminology of talking about *functions* has been preserved, even though those functions are in fact *procedures*. Threaded functions are thus similar to the threads found in Java [?] or POSIX [?]. A distinguishing characteristic of EARTH's threaded functions, however, is that they can themselves contain an additional level of (finer-grain) parallelism, called *fibers*. A fiber is an independent and *lightweight* thread of control that corresponds strictly to a segment of code *inside a threaded function*. Because the different fibers of a threaded function share the same context, *viz.*, the activation frame of their parent function, 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. The only exception is the initialization fiber (the code at the beginning of a threaded function, indicated by "`Init. fiber`" in Fig. 1) which is scheduled for execution as soon as the thread is activated. Note that being scheduled for execution does not mean the fiber gets executed immediately, as there may not necessarily be any available processor. Also note that the signals received by a fiber generally represent indications that some appropriate data dependencies have been satisfied.

- Instructions within fibers execute sequentially based on the underlying language semantics (in our case, C).
- Fibers execute in a non-preemptive and non-blocking manner. In other words, a fiber must never block because of data dependencies. These properties of fibers give rise to a *split-phase* style of programming.

EARTH's approach with threads and fibers can be seen as a trade-off between pure dataflow machines (which have an unlimited number of very fine grain contexts) and classical von Neumann architectures (which allow a limited number of heavy weight contexts). In a pure dataflow machine, each instruction is an independent thread of execution. A large number of parallel fine-grain threads can thus co-exist, but with large synchronization costs [**?**]. EARTH's approach makes it possible to obtain threads of varying granularity: a fiber can be made of a sequence (simple or complex) of instructions, a thread can be composed of one or more threads (all sharing the same context, that is, the thread's activation frame), or a program can be composed of multiple threads.

EARTH's memory model

Another important characteristic of the EARTH programming model is its underlying memory model. Following the trend in current parallel machines that focus on ensuring scalability, the EARTH model assumes a shared *global addressing space* and allows for physically distributed memory with non-uniform access time (NUMA). It is important that this assumption be made explicit since an EARTH fiber is required *never* to block (data-driven scheduling), even when a long latency operation needs to be performed. Thus, a notion of *remote* pointer, called a `GLOBAL` handle in Threaded-C, needs to be supported in order to refer to and access possibly remote locations. Furthermore, accesses to such remote locations can only be done using special split-phase instructions, in order to ensure the non-blocking property: a first fiber thus makes a request for a remote location, and then the result is received by another, distinct, fiber.

Parallel programming models

There exists a wide variety of parallel programming models. An interesting classification is that of Skillicorn and Talia, who simply define a "*model of parallel programming* [as] an interface separating high-level properties from low-level ones" [**?**, p. 128]. The various categories of models they introduce are distinguished by the various aspects which must be handled when writing parallel programs:

- Decomposition of a program into threads.
- Mapping of threads into processors.
- Communication among threads.
- Synchronization among threads.

More precisely, the various categories are defined in terms of which of these aspects must be expressed *explicitly* by the programmer. At the highest level of abstraction, there are models where nothing is explicit, not even the parallelism, for example, purely functional languages.

At the other end of the abstraction spectrum, models require the programmer to express everything explicitly, i.e., decomposition, mapping, communication and synchronization — PRAM, MPI or PVM all fall into this category. In between these two extremes, one can find models where only some aspects must be expressed explicitly, for example, in Linda, decomposition and mapping are explicit while communication and synchronization are implicit.

Inside these various categories, Skillicorn and Talia also distinguish the various models according to their degree of control over structure and communications, i.e., models in which the thread structure is dynamic vs. static and, in the latter case, models in which communication is limited.

Programming models which are lower in Skillicorn and Talia's abstraction hierarchy are generally closer to some specific execution models and can lead to more efficient programs. On the other hand such models can make programming more difficult.

The goal of the EARTH project is first and foremost to explore the architecture issues associated with fine-grain parallel multi-threaded architectures. Thus, it is important to have a programming model that makes it possible to explicitly address these various issues. In the next section, we describe the language designed to embody such a programming model, a language that falls in the "everything explicit (with dynamic thread structure)" category of Skillicorn and Talia.

The Threaded-C language

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) and the underlying memory model. The original design of Threaded-C focused on performance and was fine-tuned for the first hardware on which the model was implemented: the MANNA machine [?]. As a consequence, some features of the early versions of the language exposed details of that machine organization and architecture. This narrow focus on performance resulted in a language that was at times unwieldy to less hardware-inclined programmers (see [?] for more details).

Many of those limitations can be explained because Threaded-C was not initially designed to be used for application development. Instead, the goal was for Threaded-C to be mostly a target language for compilers. A different language, called EARTH-C [?], was designed for application programming. As it turned out, however, EARTH-C was never adopted by application programmers (see Section 6 for a more detailed discussion about EARTH-C) and Threaded-C rapidly became the common language for writing programs for EARTH machines.

Using the initial version of the Threaded-C language, various applications have been developed, for example: adaptive finite element meshes [?], computational finance [?], 2D discrete wavelet transform [?], Fourier transform [?], solution to linear equation systems using *conjugate gradient* [?], genome comparison [?]. However, experience has shown that this version of the language was far from easy to use, as confirmed by interviews and discussions made with users of the language [?].

In order to facilitate the writing of programs for the EARTH architecture, a new version of the Threaded-C language (release 2.0 [?] — the version presented in the remaining of this paper) was developed. In this revised version, we strive to preserve a narrow semantic gap

```
 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 2. Threaded-C recursive function for computing the nth Fibonacci number

between the language and the underlying architecture model, while simplifying the language and making it easier to use. We also add some language constructs to support programming on SMP clusters, where support for atomicity and mutual exclusion is needed.

An example illustrating Threaded-C (release 2.0)

Figure 2 presents a recursive function `fib`, written in Threaded-C (release 2.0), that computes the nth Fibonacci number. The keyword `THREADED` before `fib` (Line 1) indicates that parallel activations of this function — i.e., *threads* — can be created and that each such thread can itself contain finer-grain fibers.

One key characteristic of the EARTH model, apparent also in Threaded-C, is its underlying memory model. As alluded earlier, although EARTH supports a global address space with uniform addressing, it does not presume that remote locations can be accessed using ordinary load/store instructions. Thus, Threaded-C introduces the notion of a `GLOBAL` handle — a pointer to a location that *may* be in a remote memory — as well as special instructions used to transfer data to/from remote locations, e.g., `PUT_SYNC`.

In Threaded-C, subroutines are called "*functions*", even though they are not functions in the (functional programming) sense of the word. This mis-naming is especially evident in the case of `THREADED` functions that are used to represent threads that are started *asynchronously*. In other words, such functions are not *called* and do not *return* to the caller, as occur in the sequential case. Thus, functions declared as `THREADED` always have an implicit `void` return type and must return their results through reference parameters — returning a function result for a thread might make sense in a language based on explicit `fork/join`-style operations (for

example, Pthreads with its `exit` operation) but not in Threaded-C, where blocking operations are not allowed (see the end of Section 2). The argument `result` in the function `fib` presented in Fig. 2 (Line 1) is such a result parameter, while `n` simply indicates which Fibonacci number must be computed. The role of `done` is explained below.

The following additional explanations about the example of Fig. 2 will help understand the semantics of Threaded-C threads and fibers.

When an activation of `fib` is created, the initialization fiber (the code at the beginning of the function body, before Line 12 in Fig. 2) is scheduled for execution as soon as the appropriate activation frame has been created (but will get executed only when selected for execution by a processor). When the initialization fiber does start executing, it first checks (Line 5) whether recursive calls must be performed or not. There are then two possible cases for how the execution proceeds.

1. In the base (non-recursive) case (Lines 6–7), the value `1` is returned using a `PUT_SYNC` statement targeted to location `result`. This variable is a reference to a location of type `int`, possibly remote (`*GLOBAL`) since the caller may be executing on another node. When the transfer of the value `1` into location `result` is complete, a signal is sent to the synchronization slot `done`. The role of this synchronization is to indicate to the caller that the callee has finished producing its result (a data dependency has been satisfied). The current thread's job is complete and the thread terminates (Line 7), therefore the thread's activation frame can be deallocated.
2. In the recursive case (Lines 8–11), two threaded function activations (two threads) are created using the `TOKEN` statement. Independent, parallel threads can be created using either `TOKEN` or `INVOKE` statements. In our example, `TOKEN`s are used (Lines 9 and 10), and thus the run-time system (RTS) is responsible for selecting the processor on which the thread will be executed; a programmer can also use an `INVOKE` statement — for ex., `INVOKE(i, proc, ...)` — to explicitly select the processor that should run the thread. These two recursive and parallel invocations of `fib` return their respective results using distinct locations (`r1` and `r2`) but they both send a completion signal to the same slot, as indicated by the use of the same last argument expression `TO_SPTR(READY)`.

   After the creation of the two parallel activations of `fib`, the initialization fiber stops executing: in Threaded-C, the normal flow of control is obeyed until either a `TERMINATE` statement or a `FIBER` keyword is encountered. When a `TERMINATE` statement is executed (for ex., Line 7), the thread immediately terminates — it is a run-time error for a thread to terminate while there are still fibers active or already scheduled for execution. When a `FIBER` keyword is encountered (Lines 9–13), the current fiber simply stops executing, although the thread remains active since other fibers from the same thread may be ready to execute or, as is the case in this example, may be waiting for synchronization signals before they are allowed to proceed.

In the EARTH execution model, synchronization slots (also called *sync slots*) are used to receive signals and to determine when a fiber can be scheduled for execution, namely, when the slot's associated count drops to `0` indicating that its associated data dependencies have been satisfied. In other words, fibers are signaled *indirectly* through their associated sync slot. For

example, in Fig. 2, a reference to sync slot `READY` is obtained (Lines 9 and 10) using the `TO_SPTR` operator (`SPTR` = Slot PoinTeR), which returns a reference that can be transmitted to other threads and can be used in synchronization and communication instructions (`SYNC`, `PUT_SYNC`, `GET_SYNC`, `BLKMOV_SYNC`).

Threaded-C (release 2.0) allows for the implicit initialization and binding of sync slots. In the example above, the declaration of fiber `READY` (Line 13) automatically creates a sync slot named `READY` and associated with fiber `READY`. The "`<* 2 *>`" construct appearing after the fiber name (Line 13) specifies how many signals must be received before the fiber becomes enabled (ready for execution). In this case, two such signals must be received, since the results from the two recursive calls must be available before the local result can be computed and then be sent to the caller (Line 14), after which the thread can terminate (Line 15).

Admittedly, dealing with `GLOBAL` handles is not always easy. For instance, a Threaded-C program manipulating pointers and dynamic data structures might run correctly on a single node machine but, when executed on a multi-nodes machine, may fail to work if the pointer manipulations are not done using the appropriate `GLOBAL` handles and associated operations. However, clearly distinguishing between local and remote accesses is important because of Threaded-C programming and execution model: the decomposition into threads and fibers must be explicit and fibers must be defined so that they never block and get executed only when all their appropriate data dependencies have been satisfied.

Support for atomicity and mutual exclusion

Another key goal in designing the revised version of Threaded-C was to ensure that Threaded-C programs execute correctly on various implementations of the EARTH architecture, including SMP clusters where multiple processors can enjoy shared access to portions of the memory. In general, on such machines, there can be multiple fibers, all executing at the same time and accessing the same node memory, including multiple instances of fibers from the same thread. Appropriate mutual exclusion mechanisms must thus be provided. The revised language introduces two such mechanisms:

Mutually exclusive fibers: A fiber declared as `EXCLUSIVE` will always be the *only exclusive fiber* of a given thread to be executing at any given time, property which will be ensured by the run-time system (similar to Java's `synchronized` [?]).

Atomic mailboxes: This data type provides a form of non-deterministic merge operator, as typically found in dataflow models: a place where multiple messages from different sources can be merged and stored until retrieved by consumers.† Some key operations for this new data type are (see [?] for additional operations):

†Deterministic mailboxes would be convenient for parallel program debugging. Mailboxes, however, are mostly used for inter-processor synchronization and the implementation of deterministic mailboxes would be very costly (e.g. would require some tagging mechanism), because inter-processor communication delays are variable.

- `void INIT_MAILBOX(MAILBOX *mb, SLOT s)`: Allocates a mailbox on the local processor and associates with it the synchronization slot `s`. A signal is sent to `s` each time a new item arrives in the mailbox.
- `void 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` arrives in mailbox `mb`, a signal is sent to `mb`'s associated sync slot.
- `int RETRIEVE_ITEM(MAILBOX mb, void* item)`: Retrieves an (arbitrary) element from `mb` and stores it in the space indicated by `item`. The size (in bytes) of the retrieved item is also returned. Contrary to `DROP_IN`, this operation must be executed on the node where the mailbox has been allocated. If the maximum size of the item cannot be known beforehand, another operation (`RETRIEVE_ITEM_ADDR`, see [?]) can be used to dynamically allocate a buffer of the appropriate size and then return its address.

An example illustrating the use of atomic mailboxes and exclusive fibers is presented in Fig. 3, where a master process receives values from various producers and adds them together (i.e., performs a reduction process with operator "+"). In this example, the initialization fiber of the master process (`MAIN`, which must necessarily be `THREADED`) first creates a `producer` thread on each of the processors (Lines 14–15, where `NUM_NODES` indicates the number of processors currently available on the machine[‡]), sending them a reference to the mailbox `mb` which was allocated on the local processor and was bound to slot `CUMULATE_ITEM` (Line 13).

Each of the producers then transmits (Line 4) its unique identification number (`NODE_ID`) using a `DROP_IN` instruction targeted to the global reference to `mb` received as argument (Line 1). When the item reaches its destination, a signal is automatically sent to the sync slot `CUMULATE_ITEM` (Line 17), which was bound to `mb` on initialization (Line 13). Each signal sent to this sync slot triggers a *new and distinct activation* of the `CUMULATE_ITEM` fiber — this is our first example of a fiber for which multiple instances are created dynamically. The body of the fiber `CUMULATE_ITEM` then retrieves one of the items received through the mailbox (Line 20) [§] and updates the variable `total` (Line 21). In this example, the updates to variable `total` are atomic because fiber `CUMULATE_ITEM` is annotated as `EXCLUSIVE`. If multiple items arrive at the mailbox "at the same time," multiple instances of the fiber are ready to execute, but only a *single* instance at a time is allowed to proceed. Atomicity of the mailbox operations themselves is ensured by the RTS.

Each time an item is retrieved and processed by an instance of fiber `CUMULATE_ITEM`, a signal is sent to sync slot `PRINT_RESULT` (Line 22). After all items are received and processed, (one per processor, thus `NUM_NODES` as shown in Line 25), the content of `total` is printed and the program terminates.

[‡]The name "`NUM_PROCS`" might have been more appropriate, since we need to know the number of distinct processors available on the machine. However, mainly for historical reasons, "`NUM_NODES`" is used.
[§]The operation `RETRIEVE_ITEM` is a function that returns the size (number of bytes) of the retrieved item; if no item was present, `0` is returned. The specification for the atomic mailboxes do not *require* the RTS, which manages mailboxes, to retrieve the items in the order in which they were received.

```
 1  THREADED producer( MAILBOX *GLOBAL mb )
 2  {
 3    int n = NODE_ID;
 4    DROP_IN( mb, &n, sizeof(int) );
 5    TERMINATE;
 6  }
 7
 8  THREADED MAIN()
 9  {
10    MAILBOX mb;
11    int i, total = 0;
12
13    INIT_MAILBOX( &mb, CUMULATE_ITEM );
14    for( i = 0; i < NUM_NODES; i++ )
15      INVOKE( i, producer, TO_GLOBAL(&mb) );
16
17    EXCLUSIVE FIBER CUMULATE_ITEM <* 1 *> {
18      int v;
19
20      RETRIEVE_ITEM( mb, &v );
21      total += v;
22      SYNC(PRINT_RESULT);
23    }
24
25    FIBER PRINT_RESULT <* NUM_NODES *> {
26      printf( "total = %d\n", total );
27      TERMINATE;
28    }
29  }
```

Figure 3. A reduction process with multiple producers, atomic mailbox and exclusive fiber

In this example, each `producer` thread is created on a distinct processor. Therefore the use of an exclusive fiber and of an atomic mailbox leads to a program that requires a minimum number of inter-processor communications: once created, each `producer` performs a single communication using the `DROP_IN` operation.¶ Yet, all producers are allowed to proceed concurrently and the updates to `total` can be done incrementally, while the values are being received. This property would not be possible if the values were received through a fixed-size array instead of a mailbox, since all values would need to be received before their sum could be computed. On the other hand, if there was a single target location (e.g., `v`) shared by all producers, exclusive access to this location by each producer would need to be provided.

¶This communication will not necessarily be with a remote processing node, since an `INVOKE` of a `producer` is also done on processor `0`, where the `MAIN` thread always executes. Furthermore, on an SMP machine, there may be multiple processors with distinct `NODE_ID` which are part of the same processing node.

Although exclusive access could be granted through some kind of explicit user- or library-defined mutual exclusion lock, access to the lock itself by each of the producer would incur additional overhead (inter-processor communications to grab and release the lock). On the other hand, with the mailbox solution, although there is indeed a lock, it is *implicit* (i.e., handled by RTS) and, more importantly, all accesses to this lock are strictly *local* to the mailbox (i.e., requiring no inter-processor communication).

Split-phase operations and other synchronization mechanisms

The non-blocking EARTH fibers lead to the use of *split-phase* operations. In a split-phase operation, the request for some data or resource is done in a distinct phase than the reception and manipulation of that data or resource. In Threaded-C, this means that these two steps have to be performed by two distinct fibers.

The most basic example of a split-phase operation is `GET_SYNC`, an operation used to retrieve a value from a possibly remote location (the last argument can also be a local sync slot instead of a remote one, in which case the compiler will provide the appropriate mapping):

- `void GET_SYNC( T *GLOBAL src, T *GLOBAL dest, SPTR s )`

When `GET_SYNC` is executed, a value of type `T` (any type, including `struct`s) is transferred from the source address `src` into the destination address `dest` and, when the transfer is completed, the sync slot `s` is signaled. An important feature of the split-phase operation is that the `GET_SYNC` instruction completes (almost) *immediately*, not when the transfer has been performed. For instance, suppose that a thread needs to retrieve the content of some remote locations `x` and `y` in order to perform some work on local copies of these locations (`my_x` and `my_y`). This data transfer is expressed by the following explicit split-phase operations (the two requests can proceed in parallel):

```
GET_SYNC( x, my_x, XY_RECEIVED );
GET_SYNC( y, my_y, XY_RECEIVED );
/* my_x and my_y are *not* yet available. */

FIBER XY_RECEIVED <* 2 *> {
  /* Further work on my_x and my_y is now allowed. */
  ...
}
```

Threaded-C dataflow-style communication and synchronization operations together with exclusive fibers and atomic mailboxes have been used to develop library modules that define and implement other synchronization mechanisms such as locks, semaphores, I-structures [**?**], (uni- and bi-directional) communication channels, parallel reduction boxes.[‖] For example,

[‖]A number of these library modules can be found at the following URL:

`http://www.capsl.udel.edu/EARTH/LIBRARY-DOC/`

grabbing a (*split-phase*) lock in order to define a critical section would look as follows, where exclusive access is ensured only when a signal has been received by slot `LOCK_OBTAINED`, not immediately after the `LOCK_SYNC` has been executed:

```
LOCK_SYNC( lock, TO_SPTR(LOCK_OBTAINED) );
...

FIBER LOCK_OBTAINED <* 1 *> {
  /* Begin critical section. */
  ...
  UNLOCK( lock );
  /* End critical section. */
}
```

The general strategy for implementing split-phase locks, as well as many of the other synchronization mechanisms, is based on an approach similar to the so-called *active monitors* described by Andrews [?, Chap. 7]. A distinct thread — an instance of a `lock_handler` function — is created each time a new lock is allocated and initialized (using `INIT_LOCK`). A call to a lock operation is first handled by a proxy (since the lock can be on a remote processor) that sends an appropriate request for manipulating the lock to the associated lock handler (the thread created for that lock). Such a request can be a simple signal sent through a regular sync slot (e.g., `UNLOCK`) or a more complex message sent through a mailbox (e.g., `LOCK`). Arrival of the request will then trigger an appropriate fiber in the `lock_handler` thread. Atomic manipulation of the lock is ensured by defining all the lock handler's fibers as `EXCLUSIVE`. For a more detailed presentation of the implementation of split-phase locks, the reader can consult the Threaded-C (release 2.0) reference manual [?]. In the next section, we discuss the architecture model and the runtime system that implements the Threaded-C primitive operations described in this section.

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

Figure 4 shows the organization of the EARTH architecture model. In this model, processing nodes are interconnected via a network. 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. The EUs and the SU communicate with each other through a ready queue (RQ) and an event queue (EQ). The ready queue holds fibers that are ready for execution, i.e., fibers that have received all their appropriate synchronization signals. A fiber in the ready queue is waiting for an EU to become available. In the SMP implementation described in this paper, the functions of the EU are implemented by one or multiple Execution Modules (EM) as indicated in the box on the top left of the figure. The event queue contains events yet to be handled by the SU. For instance, events in this queue may include a signal to be sent to a local sync slot, or a request to initiate a transfer to/from a remote memory location node.

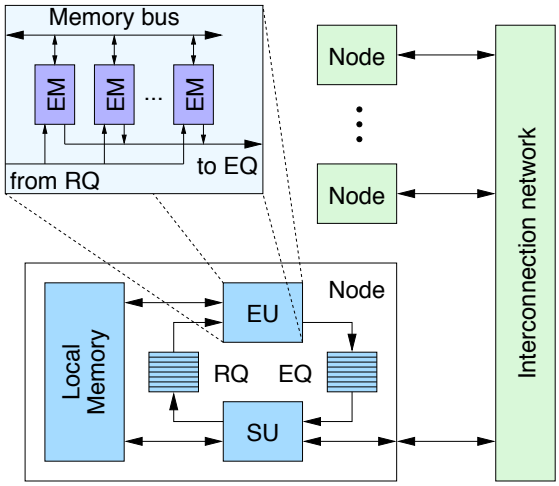


Figure 4. EARTH architecture

Because fibers correspond to sequences of instructions and are non-preemptive, the instructions that constitute a fiber can be executed using a regular instruction pipeline. The SU takes care of all synchronization tasks. The technology used to implement the SU provides a trade-off between speed and portability for implementations of EARTH. A custom hardware SU tightly coupled with the network interface and with the node's CPUs would provide the most efficient implementation of EARTH, but would also be the least portable. An implementation of the SU using Commercial Off-The-Shelf (COTS) hardware, but with access to the software that controls the flow of messages in the inter-node network cards — the MANNA and the SP-2 implementations of EARTH follow this model [?] — provides less efficiency but is more portable because only a small part of the EARTH RTS needs to be re-written when the network interface changes. The most portable solution — used in the implementation described in this paper — is a software implementation of the SU's functionality using standard network interfaces. This solution also avoids the dedication of a CPU to exclusively execute the SU functions.

When standard COTS processors are used to build an EARTH machine, the runtime system (RTS) handles the tasks of creating threads, scheduling fibers, and handling network communication. In the next section, we present our new design for this RTS.

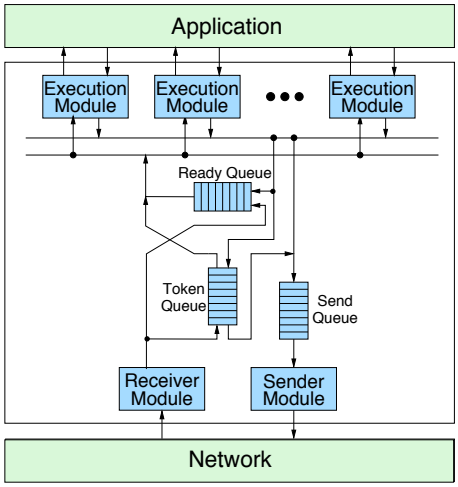


Figure 5. Multiple execution module EARTH runtime system

4. The Design of the New Runtime System

Our goal was 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 5. The Execution Modules (EM) execute 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. Whenever idle, an execution module fetches a fiber from the ready queue (RQ) or a token from the Token Queue (TQ). When a token is fetched it is expanded in its component fibers and a thread frame is allocated for the thread encoded in the token. Once the token is expanded, it can no longer migrate to be executed in other processing nodes. The Receiver Module (RM) (resp. Sender Module, SM) 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. Notice that the tokens may be added to and removed from both ends of the Token Queue. The idea is that new work generated by the local execution module is placed in one end of the TQ, and is more likely to be executed by one of the local EMs. Meanwhile, at the other end of the TQ, the automatic load balancer will either place newly arrived work from other processing elements or steal tokens to send to other processor elements. The Ready

Queue (RQ) contains fibers that must be executed locally, and the Sender Queue (SQ) contains the outgoing messages.

The number of EMs per SMP node can be configured when the RTS is generated. Machines that have more processors in a single SMP node can benefit from concurrent execution within a processing node by allowing multiple EMs to run on different processors. When multiple EMs are active, each EM has its respective Ready Queue, but all the EMs share a Token Queue and a Send Queue. All the modules are implemented as POSIX threads (pthreads), and therefore access the same memory space. Intra-node communication is accomplished simply and efficiently through memory reads and writes.

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 trade-off between polling and interrupts, blocking vs. non-blocking I/O, and the potential deadlocks in an RTS.

The convenience of 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.

The details of the socket API are described elsewhere [?]. In Figure 6 we illustrate how the socket API and protocol is used to implement `PUT_SYNC`, an important EARTH primitive, in our implementation of the EARTH runtime system. Assume that the following call to `PUT_SYNC` is executed on node `n1`. The parameter `k` is an integer, `dest` is a global handle referring to a location on node `n2` (assume `n1` $\neq$ `n2`), and `lslot` is a reference to a local sync slot (on node `n1`):

```
PUT_SYNC(k, dest, lslot);
```

This operation transmits the value `k` to the remote location `dest` and sends a synchronization signal to `lslot` when the transfer is complete. The runtime system determines that the destination of the `PUT_SYNC` is in a remote node and places a message in a queue (`etc_enqueue_send()`). After a `write()` operation in the sending node and a corresponding `read()` operation in the receiving node, the message is decoded (`hdl_data_msg()`), the value of `k` is written in the specified address, and a sync operation is generated (`etc_sync()`). This sync operation generates a message that is transmitted back to `n1`, decoded, and the sync slot specified `lslot` is decremented(`hdl_sync()`).

Our implementation of the EARTH RTS uses the blocking mode of access to sockets. Using this mode, we can avoid polling, and we can also issue blocking calls to `select()`. Such calls to `select()` only return when incoming messages have arrived in a socket's receive buffer.

To poll or not to poll

We know three alternatives to implement the communication between the runtime system and the network interface: interrupts, polling, and polling-watchdog. Interrupts are usually not

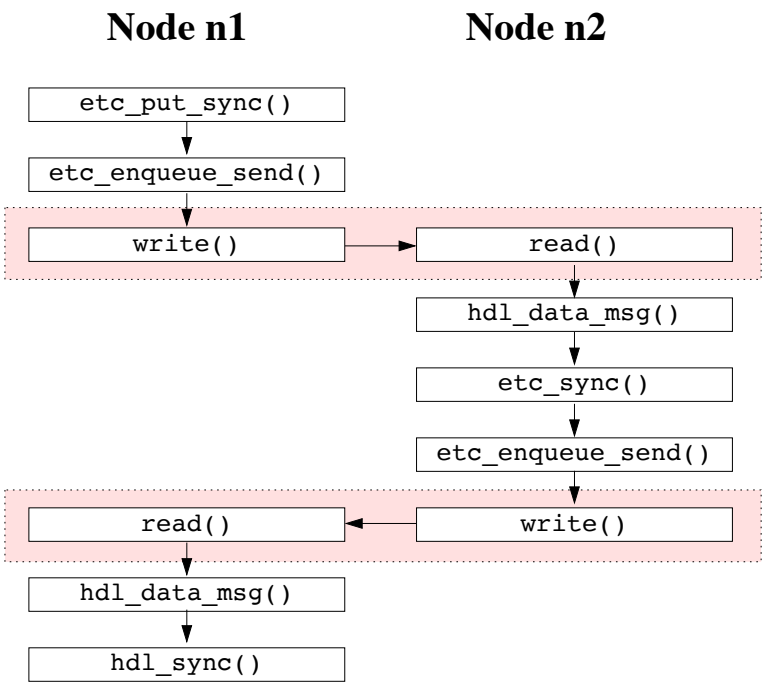


Figure 6. How the socket functions are used to implement `PUT_SYNC`

desirable in a multi-threading system because an interruption of the running thread leads 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 [**?**], 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 is started; if the message is not handled within a given amount of time, the network interface interrupts the runtime system. The advantage of the polling-watchdog approach 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, we must be able to program the network interface to define an appropriate time-out mechanism. Such modifications to the network interface were implemented in earlier versions of the EARTH runtime system, but they made those systems less portable.

Since Unix sockets are used as our inter-processing node communication mechanism, a polling approach would use the `select()` system call, requiring the kernel to perform a linear search on its socket structures to identify which socket has an incoming message. Thus, in a

large cluster with many open sockets polling can become very expensive (between 2,500–15,000 processing cycles for a single poll operation).

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 process that is currently running. 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.

Blocking vs. non-blocking I/O

After the runtime system has been notified that a message arrived, it needs to transfer the message's content, using a `read()` operation, from the socket buffer in the kernel space into a buffer in the 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 behaviors are affected by the use of blocking or non-blocking I/O. Such effects are observed when a read requests more bytes than are currently available in the socket buffer, or when the buffer overflows because of a write operation. In a non-blocking I/O system, any read or write operation returns 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 [?]. Because modern processors are much faster than 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 a single thread of execution, with a single program counter, alternates between executing EARTH threads and handling network activity, a large number of CPU cycles are likely to be wasted. Any time a socket operation blocks, the CPU sits idle until the socket operation is able to complete.

Blocking socket access is much simpler to implement than non-blocking I/O. In our system we leverage the simplicity of blocking I/O, but avoid wasting CPU cycles by splitting the networking functionality of the runtime system into two separate POSIX threads, as shown in Figure 5. We name these two networking threads the "Sender Module" and the "Receiver Module", to avoid confusion resulting from overloading the use of the term "thread" (Threaded-C threads vs. POSIX threads used in the RTS).

When blocking I/O is used, a potential deadlock condition may arise when a socket *send buffer* on one end of a link becomes full. More precisely, both nodes might become blocked in a `write()` operation to their send buffer, both waiting for the other node to read from the corresponding receive buffer in order to allow communication to proceed. If both the writing

and reading tasks are handled by a single process, then a deadlock situation will arise.** More general deadlock situations can also occur during the execution of multi-threaded programs that have complex cycles of inter-node dependencies.

Our implementation of the two separate modules avoids the deadlock problem. When the sender module blocks, the host operating system will switch to another available POSIX thread, in this case the receiver module, and read any incoming data that is there, thus allowing the system to make progress. If the scheduling is fair with respect to the receiver module, this decomposition into separate modules will avoid deadlock situations, even if the execution module is allowed to proceed when the sender module blocks.

5. Experimental Results

The experimental platform

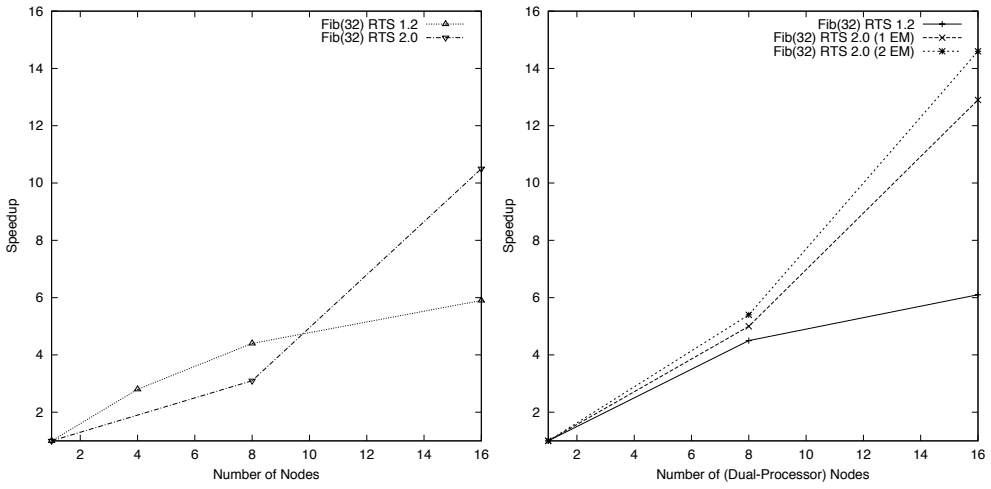
We installed the EARTH runtime system described in this paper on two Beowulf clusters: "Earthquake" operated by the Computer Architecture and Parallel Systems Laboratory (CAPSL) at the University of Delaware, and "Ecgtheow" operated by the Computational Science and Engineering program at Michigan Technological University and sponsored by the NASA High Performance Computing and Communications Office (HPCC) for the Earth and Space Sciences (ESS) project. Earthquake has 16 500MHz Pentium III processor nodes with 128MB of RAM. Ecgtheow has 64 nodes, each with dual 200 MHz 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.

In order to evaluate the influence of the runtime system design on the performance of the EARTH architecture in an SMP cluster, we ran two versions of the runtime system: RTS 1.2 and RTS 2.0. The 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) for each processing node in a single module. By contrast, the RTS 2.0 uses interrupts to interface with the network, implements blocking sockets, and separates the execution of fibers, the sending, and the receiving activities of the network into three separate modules.

Test programs

We used three programs to evaluate our implementation of the runtime system: (1) a recursive implementation of Fibonacci in which each non-base call to the Fibonacci function generates two distinct recursive calls (see Figure 2, p. 5, for the Threaded-C version of this procedure);

**This situation did occur in an earlier implementation of the RTS. This problem was quite difficult to identify and fix since it was hard to reproduce because of its inherent time-dependent behavior.



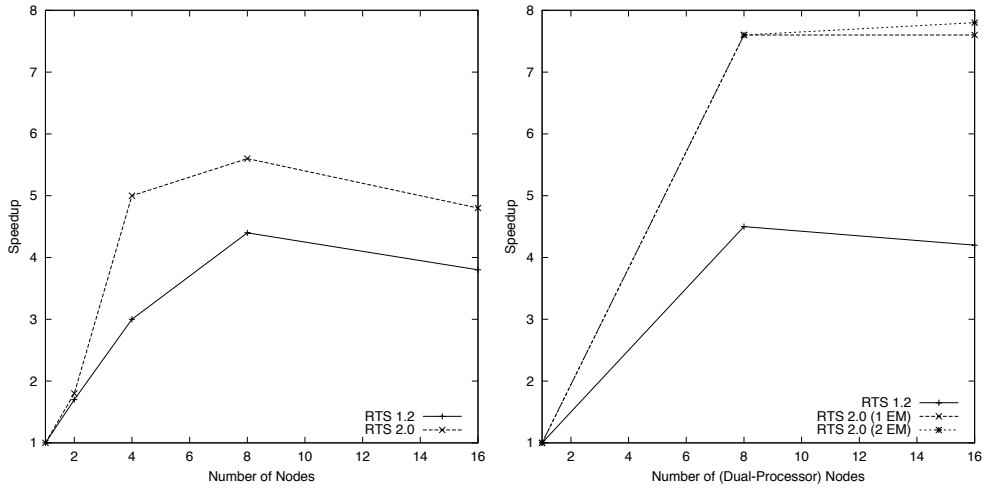
(a) (b)

Figure 7. Speedup curves for Fibonacci: (a) On Earthquake (b) on Ecgtheow

(2) a recursive, non-throttled implementation of N-queens;†† and (3) ATGC (Another Tool for Genome Comparison), a multi-threaded implementation of the Smith-Waterman algorithm, a dynamic programming algorithm, for sequence comparison [**?**]. We report results for 16 single processor nodes for all three benchmarks on Earthquake. On Ecgtheow we report curves for 16 dual processor nodes for Fibonacci and N-queens and the results for 60 dual processor nodes for ATGC. We do not report results in 60 nodes for all benchmarks in Ecgtheow because the use of all the 60 nodes requires special coordination with other users of that cluster.

Figure 7 presents the speedup curves for runs of `fib(32)` (a recursion with 4.3 billion leaves) on Ecgtheow and Earthquake under both the RTS 1.2 and the RTS 2.0. The recursive Fibonacci 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 implements a single execution module in each processing node, while RTS 2.0 (2 EM) implements two execution modules per processing node.

††We say that a recursive application is "throttled" when after a certain number of recursions the program executes a non-recursive version of the code. The application is "non-throttled" when the recursion proceeds all the way to the base case. If each function invocation is a new thread, a non-throttled recursive application is a good test for the ability of an architecture to handle a large number of threads or processes.

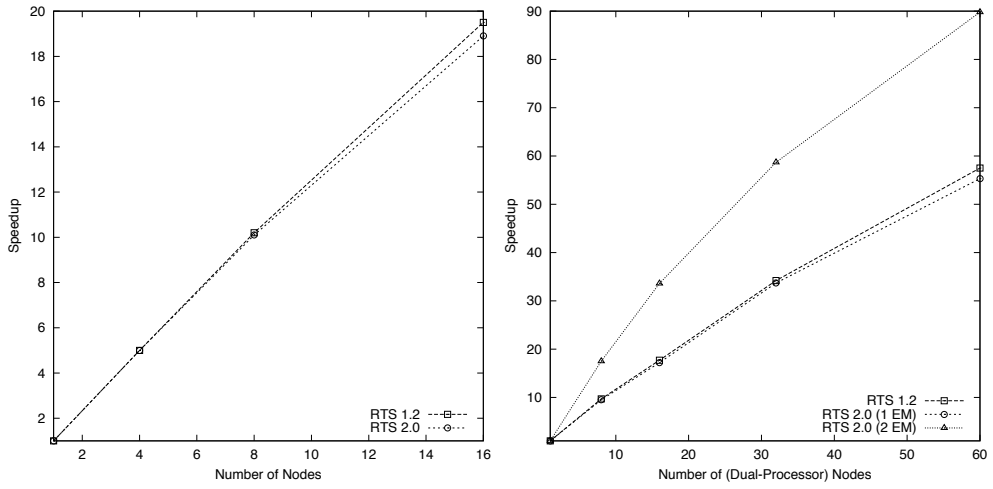


(a) (b)

Figure 8. Speedup curves for N-Queens(12): (a) On Earthquake (b) on Ecgtheow

For clusters with more than 10 processing nodes, the RTS 2.0 significantly outperforms the RTS 1.2 in both machines. When two EMs are used in Ecgtheow, the RTS 2.0 delivers a speedup of 15 in 16 dual-processor nodes, compared with a speedup of only 10.5 for the RTS 1.2. On Earthquake when only eight single-processor nodes are used, the RTS 2.0 under-performs the RTS 1.2. This is because, with a single processor per node, the cost of switching between the multiple modules of the 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 decline in speedup for machines with more than eight processors is explained by the small amount of work in each thread in these applications. Thus using more processors causes the internode communication and the load balancing activities to dominate the time consumption in relation to the actual execution of threads.

The goal of the N-queens program, a recursive algorithm representative of some typical highly parallel applications, is to determine the number of ways in which n queens may be placed on an $n \times n$ chess-board so that no queen is in a position to attack another. Figure 8 shows the speedup curves for N-queens on a 12×12 board. Again, the RTS 2.0 significantly outperforms the RTS 1.2. In Figure 8(a) we experience a true superlinear speedup over the sequential version of the code when four of the single-processor nodes of Earthquake are used. This superlinear speedup can be explained by the availability of higher memory bandwidth in the four processor machine when compared with the single processor one. In Figure 8(b), there is no superlinear speedup (apparent or real) because a machine with 8 dual-processor nodes has 16 processors.



(a) (b)

Figure 9. Speedup curves for ATGC: (a) On Earthquake (b) on Ecgtheow

Also important to note is the decline in speedup from 8 to 16 nodes with both versions of the runtime system. The non-throttled version of N-queens executes the recurrence until the end, and thus a large number of small threads are generated. These threads are distributed on the processing nodes in the system by an automatic load balancer. With a larger number of nodes in the system, the overhead of transferring threads to more nodes is higher than the gain obtained from the availability of more processors. This situation would be less prominent in machines where the difference between the inter-node network speed and the CPU speed in each node is not as big as in the case of Earthquake.

Much better speedups for N-queens(12) on EARTH can be obtained when the program is throttled at an adequate level and stops generating new threads. In this study, we do not throttle N-queens because our goal is to assess how well the RTS handles high volumes of tokens.

The third benchmark — ATGC — is a dynamic programming algorithm for DNA sequence comparison. We used this program to compare two random DNA sequences, both of size 40K base pairs. In the graph of Figure 9, the speedup curves for the RTS 1.2 and the RTS 2.0 using one EM in each processing node are very similar. ATGC is an application which is far more CPU intensive than network intensive. Thus, the advantage of the new design of the RTS 2.0 is made more evident when a second EM is activated. The RTS 2.0 is able to fully utilize both processors in every SMP node of Ecgtheow, which results in nearly double the speedup for ATGC. Notice that when executing on 60 nodes, Ecgtheow is using 120 processors. Thus, the speedup of 90 in Figure 9(b) is in fact sub-linear.

6. Related Work

The EARTH model has its origin in the argument-fetching dataflow model, a dataflow model *without flow of data* [?]. EARTH also has been influenced by early work in multi-threading parallel architectures [?]. Earlier implementations of the EARTH system are described in [?, ?, ?]. The ATGC program is described in [?, ?].

The EARTH runtime system implements an extensive set of elaborate dynamic load balancers to enable the automatic distribution of the computation load when the programmer uses the `TOKEN` construct in Threaded-C [?]. Amaral *et al.* [?] implemented I-structures [?] for EARTH and demonstrated that I-structure caches can be effective. This is especially true for cluster implementations where there is a significant gap between the network latencies and the processor speed.

A programming language initially associated with EARTH is EARTH-C [?, ?]. In EARTH-C, parallelism is expressed using high-level constructs (such as `forall` and sequential vs. parallel instruction groups), and data can be declared as private or shared. The compiler, which generates Threaded-C code, is in charge of identifying the fibers boundaries and the split-phase transactions required to access potentially remote data. Because of the semantic gap between this level of parallelism expression and the distributed memory machines in which the EARTH model was implemented, compilers had limited success when partitioning EARTH-C into threads and fibers for Threaded-C. A major roadblock was the absence of an efficient alias analysis to determine which memory references could not be shared, and thus would not need a split-phase transaction. Recent papers made progress on both fronts: improved alias analysis [?], and better thread partitioning algorithms [?]. However, it was too late in the history of EARTH-C to reverse the trend of slow adoption of the language, which is why all major application development was done in Threaded-C.

There are a number of projects relevant to our research. Like EARTH, Cilk is a C-based multi-threaded language and runtime system [?]. However, in its initial design, Cilk was targeted exclusively toward 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 [?]. Cilk-NOW is an implementation of Cilk for networks of workstations [?, ?]. 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 multi-threaded language that can express parallelism at varying levels of granularity, efficiently supporting programs requiring irregular fine-grain parallelism.

Split-C [?] and UPC [?] are two other languages, also based on C, that support some form of split-phase programming. In Split-C and UPC, however, the thread/fiber hierarchy does not exist. Also, in both languages, a non-local access does not necessarily require an explicit communication. This can be an advantage in terms of programming model, but does not match the EARTH model of fibers (non-blocking, non-preemptive).

Another language whose roots are from dataflow is pH [?], a (mostly-)pure functional programming language. More precisely, pH is a parallel and non-lazy version of Haskell [?] with a number of non-functional extensions (I-structures and M-structures). Contrary to Threaded-C, pH is thus an *implicitly* parallel language, where the programmer has no control over the communications and the decomposition into threads and fibers.

MPI is a standard interface for the message passing paradigm that seeks to combine the most attractive features of existing message passing systems [?]. 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. On the other hand, MPI provides a rich set of operations for global communication, e.g., broadcast, scatter, and gather [?].

Some of the fundamental ideas in EARTH/Threaded-C appear in the *Filaments* system implemented by Freeh *et al.* [?]. The *Filaments* system also implements a runtime system to execute fine-grain threads, relies on the programmer to implement thread partitioning and implements dynamic load balancing in a runtime system. The *Filaments* system was designed for shared memory machines only and, thus, does not provide the mechanisms for inter-node communication available in EARTH.

An example of a multi-threaded runtime system for commercial off-the-shelf SMP machines is presented in the implementation of the Superthreaded Architecture (SA) by Kazi and Lilja [?]. Contrary to the EARTH effort, the SA does not introduce an explicitly threaded language, but instead attempts to do automatic thread partitioning and extraction of parallelism, although it seems to have had limited success in this effort. Currently, most of the parallelization is performed "by hand," and thus is limited to fairly regular parallel programs. The implementation of SA on an SMP machine is offered as an alternative strategy for simulating and testing the SA execution model, not as a high performance system on its own right.

7. Conclusion

This paper has presented the new design of the runtime system for the EARTH multi-threaded architecture, together with the revised version of the Threaded-C language used to write programs for this architecture. The intended target machines for this new RTS are modern multi-node systems with multiple processors per node (SMP clusters). We designed the RTS with the goal of being portable, yet making it possible to benefit efficiently from the power of multiple processors per node. In order to do this, our RTS implementation uses multiple threads of execution, which also precludes deadlock situations.

On the language side, current work is being done to further improve the Threaded-C language, yet preserve a narrow semantic gap that will ensure that programmers still have full control over the granularity of fibers and over synchronization and communication. For example, a notion of fiber with arguments has recently been introduced and experimented with (using a prototype pre-processor that extends the Threaded-C language [?]), allowing the flow of data to be made more explicit and further simplifying the specification of synchronization

constraints associated with fibers. Other extensions are still under investigation, for example, allowing fibers with multiple level of priorities.

ACKNOWLEDGEMENTS

The authors would like to thank current and former members of CAPSL at the University of Delaware for valuable exchange of ideas. Special thanks to Kevin Theobald for the N-queens code, and to Juan del Cuvillo and Wellington Martins for the ATGC code. Thanks to Phil Merkey and Dan Becker for interesting discussions about the RTS design, and for Phil Merkey for making Ecgtheow available. Thanks to the anonymous reviewers for very insightful comments and questions. The authors are grateful to 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. Tremblay and Amaral are supported by NSERC grants from Canada.