University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Overview of the Threaded-C Language

Kevin B. Theobald†
José Nelson Amaral
Gerd Heber
Olivier Maquelin†
Xinan Tang
Guang R. Gao

CAPSL Technical Memo 19
March 16, 1998

Copyright © 1998 CAPSL at the University of Delaware

(The names of all authors, except the first and last, are in alphabetical order)

†School of Computer Science, McGill University, Montréal, Québec, Canada

Abstract

This document describes a *portable* implementation of the Threaded-C language for EARTH. First, we briefly review the EARTH programming model. Then we present an overview of the extensions to the C language that implement this model in Threaded-C. We use a set of small examples to acquaint the reader with this new programming language and style. This tutorial is followed by two detailed examples, one a demonstration of mutual exclusion implemented with the currently available constructs in Threaded-C, and the other a complete program computing a one-dimensional FFT. The appendix contains a list of all Threaded-C commands and their usage.

Contents

1 Introduction **1**

2 EARTH Implementations and Threaded-C **2**
2.1 Variations of Threaded-C 2

3 The EARTH Programming Model **4**
3.1 EARTH Architecture 4
3.2 Threads 5
3.2.1 Synchronization Slots 6
3.2.2 Thread Scheduling 6
3.3 Threaded Functions 7

4 Introduction to Threaded-C **8**
4.1 The MAIN Function and Regular C Functions 8
4.2 Calling Functions on other Nodes 9
4.3 Synchronization Between Functions 10
4.3.1 Local and Remote Synchronization 12
4.3.2 Spawning Threads 13
4.4 Load Balancing 13
4.5 Transferring Data 14
4.5.1 Simple Data Synchronization 15
4.5.2 Fetching Data with Synchronization 17
4.5.3 Block Moves 17

5 Advanced Features of Threaded-C **22**
5.1 Variables in Global Scope and Memory Allocation 22
5.2 Polling 24

6 Examples of Threaded-C Programming **26**
6.1 Mutual Exclusion in Threaded-C 26
6.2 Example: One-Dimensional Fast Fourier Transform 30

7 Differences Between Threaded-C and ANSI C **34**

8 Conclusion **34**

A EARTH Primitives in Threaded-C **37**
A.1 Threads and Functions 37
A.2 Thread Synchronization 38
A.3 Data Transfer Primitives 39
A.4 Global Address Support 40

List of Figures

1 EARTH architecture 4
2 Synchronization slot format 6
3 Stack and activation frames 7
4 A Simple Threaded-C Function 9
5 Hello World on All Nodes 10
6 Safe Hello World 12
7 Fibonacci Code Structure 15
8 Recursive Fibonacci 16
9 Recursion in N-Queens 19
10 Data Transfers and Synchronization in the N-Queens Solution 19
11 Recursive N-Queens Code 20
12 N-Queens Checking Function and MAIN Routine 21
13 Function to Replicate Table 22
14 Allocate and Deallocate with Global-Scope Variables 23
15 MAIN Routine Calling Allocate and Deallocate 24
16 Allocate and Deallocate without Global-Scope Variables 25
17 MAIN Function Using Global-Free Allocate and Deallocate 26
18 Mutual Exclusion – MAIN Routine 28
19 Mutual Exclusion – Produce Value 29
20 Mutual Exclusion – Update Value 29
21 Mutual Exclusion – Initialize and Print 30
22 FFT Code 33
23 Examples of Illegal Use 34

1 Introduction

In multithreaded systems based on the EARTH execution model [4, 5, 6], application programs may be written in either EARTH-C or Threaded-C. Both languages extend standard C with constructs for parallelism. The difference between them is that Threaded-C code is explicitly partitioned into threads, while EARTH-C provides constructs for expressing parallelism but leaves thread partitioning to the compiler. Threaded-C code is generally harder to write than equivalent EARTH-C code, but is more efficient. This report gives an overview of one version of Threaded-C.

Several dialects of Threaded-C have been written. The first version, written specifically for the MANNA implementation of EARTH, was tuned to the MANNA architecture, and its compiler could only generate code for the MANNA [8]. Several features of "EARTH-MANNA Threaded-C" made it unsuitable for larger architectures such as high-end IBM SP-2 machines. In particular, EARTH-MANNA Threaded-C assumes that regular pointers can cover the entire global address space, which is not the case in a larger SP-2 [2]. Therefore, a second version of Threaded-C was written to be portable across all platforms, although this version has a slightly more complex syntax.

This portable version of Threaded-C is available on several platforms. The most generic implementation can run on clusters of workstations supporting TCP/IP, such as the Beowulf (Linux-based PC's with fast Ethernet). Also, several implementations have been tuned to specific platforms, delivering better performance. At the time of this writing, the platforms supported include the IBM SP-2 and SP-3, the MANNA, and a network of Sun workstations connected with Myrinet. It is also possible to run Threaded-C code sequentially on a Sun or RS6000 workstation.

This document describes this portable version of Threaded-C. The next two sections review the salient features of EARTH and Threaded-C (these have been covered in prior documents). The first section covers the history and evolution of Threaded-C and discusses current directions. Section 3 gives an overview of the EARTH program execution model, concentrating on features relevant to Threaded-C.

The remainder of this report is devoted to Threaded-C. Section 4 is an introduction to the portable Threaded-C language, using examples to illustrate its primary features. The section that follows discusses a few advanced features of the language. Section 6 shows some slightly larger examples of Threaded-C code, one demonstrating computation using mutual exclusion and the other computing a one-dimensional FFT. Programmers should take careful note of Section 7, which briefly mentions a few standard C features not allowed in Threaded-C. The final section lists materials for further information on EARTH and Threaded-C. For convenience we also include a complete list of EARTH primitives in Appendix A.

2 EARTH Implementations and Threaded-C

EARTH was first conceived as a set of extensions for off-the-shelf microprocessors to support multithreading.[1] These extensions could be implemented in a *Synchronization Unit* attached to the main *Execution Unit*, and EARTH operations in this chip could be triggered by executing certain native instructions in the EU, e.g., using memory-mapped I/O. Executable applications code would consist of threads written in the native EU machine language, including the special instructions for invoking the EARTH operations in the SU.

Since the emphasis is on multithreading and not on a particular processor architecture, the EU processor is not specified. The SU hardware is also left open; only its capabilities are dictated. This allows great flexibility in the design of an EARTH system. The first implementation of EARTH was on MANNA [1], a multiprocessor consisting of dual-processor nodes. In each MANNA node, one processor performs the EU's task of executing threads, while the other processor emulates the specified behavior of the SU. Other platforms can be used for EARTH as well, including machines with single-processor nodes (in which case the single processor has to perform both EU and SU functions).

A uniform interface was needed so that applications could be written for EARTH independent of the EARTH implementation. This led to the development of Threaded-C. Threaded-C is ANSI-standard C with extensions corresponding to the EARTH multithreading operations. A C compiler, written for the EU processor, translates the regular C into native EU instructions, with provisions made for converting the Threaded-C extensions. The Threaded-C language may be used as a user-level, explicitly-threaded programming language, or may be the compilation target of a non-threaded language, such as EARTH-C [3], or ordinary sequential C.

2.1 Variations of Threaded-C

The EARTH operations [4, 5, 6] as originally specified are frame-based. That is, each operation has arguments specifying a frame pointer and one or more offsets from that pointer, e.g., to sync slots or local variables. During the development of the first implementation, it was decided that a more efficient mapping to most processors could be made by using direct pointers rather than frame pointer/offset pairs. Therefore, all versions of Threaded-C to date use pointers directly to specific locations.

Threaded-C was first implemented on the MANNA platform. This platform was very flexible and coding could be done at a very low level. Even the on-chip memory-management unit could be manipulated, something normally restricted to the OS kernel. Thus, a highly efficient run-time system was developed to emulate the SU functions on one of the CPU's in each node. A complex post-processor was developed as part of the Threaded-C compiler to transform the code generated by a commercial sequential compiler into a form suitable for multithreaded execution.

[1] It was originally called MTA (Multi-Threaded Architecture) but was changed to EARTH to avoid confusion with another multiprocessor by the same name.

Unfortunately, moving from EARTH-MANNA to other platforms, such as a generic network of workstations, forced us to make some changes to the Threaded-C language. These changes are almost entirely due to limitations in the representation of global addresses in EARTH-MANNA Threaded-C. In EARTH-MANNA it is possible to reference the entire global address space with 32 bits (the largest MANNA implemented has only 1.28Gbytes of memory). The MMU on each MANNA processor can be programmed to map global addresses to local addresses efficiently. Thus, the same pointer can be used for both local and remote access to a memory location. However, some larger machines, such as the SP-2, can have more memory than will fit into the address space of a single processor (4Gbytes). Regular pointers won't be big enough to accommodate global addresses. Furthermore, most parallel machines don't give user-generated code access to the MMU.

Therefore, a new data type had to be created to specify global pointers. The new type specifies both a processor-node number and an address within that node. It is necessary to distinguish between global and local addresses explicitly, for the EU may only access *local* addresses; the EU must send global addresses to the SU for processing. This distinction was achieved by introducing a GLOBAL type qualifier for pointers and by providing explicit conversion operations between local and global pointers.

A second problem developed in trying to run EARTH code on a generic system. EARTH assumes there is a separate SU unit to take care of synchronization and communications tasks. In particular, the SU handles all incoming messages immediately, sparing the EU this burden. Most multiprocessors, however, don't have a second processor, and on these machines, it is necessary to perform the SU functions on the same processor that runs the application threads. How are incoming messages handled in this case? Experiments with a single-processor-node implementation of EARTH [7] found that interrupting the EU each time a message comes in imposes too many interrupt overheads on the processor, while polling the input queues only at thread boundaries could cause other processors to wait too long for their requests to be handled. For this reason, a POLL instruction was added, to suggest places for polling the input queues.

After these changes were made to Threaded-C, the MANNA implementation was modified to accept the new syntax without sacrificing efficiency. Apart from some minor restrictions, programs written for the portable version can run unchanged on MANNA. (This is not the case for programs originally written for EARTH-MANNA, however, which need to be modified before they can run correctly on the SP-2.)

A new version of Threaded-C is currently in the planning stage. It will be based on frames and frame pointers, and its commands will closely match the original EARTH operations. This will eliminate the problem of global addresses since frame pointers will take fewer bits than arbitrary addresses. It will also incorporate new features designed to simplify the writing of Threaded-C code, based on feedback from programmers who have written large applications in Threaded-C, and support for concurrency-control features found in some other parallel systems.

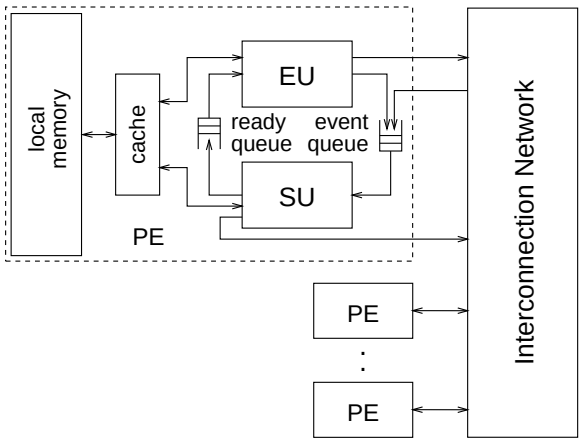

Figure 1: EARTH architecture

3 The EARTH Programming Model

In the EARTH programming model, threads are sequences of instructions belonging to an enclosing function. Threads always run to completion – they are non-preemptive. Synchronization mechanisms are used to determine when threads become executable (or ready). Although it is possible to spawn a thread explicitly, in most cases a thread starts executing when a specified *synchronization slot* counter reaches zero. A synchronization slot counter is decremented each time a synchronization signal is received. In a typical program, such a signal is received when some data becomes available. Besides the counter, a synchronization slot holds the identification number, or *thread id*, of the thread that is to be started when the counter reaches zero. This mechanism permits the implementation of dataflow-like firing rules for threads (a thread is enabled as soon as all data it will use is available).

3.1 EARTH Architecture

An EARTH computer consists of a set of EARTH nodes connected by a communications network.[2] Each EARTH node has an *Execution Unit* (EU) and a *Synchronization Unit* (SU) linked to each other by queues (see Figure 1). The EU executes active threads, and the SU handles the synchronization and scheduling of threads and communication with remote processors.

This division allows the implementation of multithreading architectures with off-the-shelf microprocessors mass-produced for uniprocessor workstations [6]. The EU is expected to be a conventional microprocessor executing threads sequentially.[3] The SU performs specialized tasks and is relatively simple compared to the EU. Thus, the SU can be implemented in a small

[2]The EARTH model does not specify the network's topology.

[3]More precisely, the EU executes threads according to their *sequential semantics*. Naturally, such a processor could take advantage of conventional techniques for speeding up sequential threads, such as out-of-order execution and branch prediction.

ASIC chip. The two queues connecting the EU and SU may be in separate hardware or may be part of the EU and/or SU.

The function of the queues shown in Figure 1 is to buffer the communication between the EU and SU. The *ready queue*, written by the SU and read by the EU, contains a set of threads which are ready to be executed. The EU fetches a thread from the ready queue whenever the EU is ready to begin executing a new thread. The *event queue*, written by the EU and read by the SU, contains requests for synchronization events and remote memory accesses, generated by the EU. The SU reads and processes these requests as fast as it is able. Request from the EU for remote data can go directly to the network or go through the local SU; implementation constraints will determine the best mechanism, so this is not defined in the model.

To assure flexibility, the EARTH model does not specify a particular instruction set. Instead, ordinary arithmetic and memory operations use whatever instructions are native to the processor(s) serving as the EU. The EARTH model specifies a set of EARTH operations for synchronization and communication. These operations are mapped to native EU instructions according to the needs of the specific architecture. For instance, on a machine with ASIC SU chips, the EU EARTH instructions would most likely be converted to loads and stores from/to memory-mapped addresses which would be recognized and intercepted by the SU hardware.

To maximize portability, the EARTH model makes minimal assumptions about memory addressing and sharing. An EARTH multiprocessor is assumed to be a distributed memory machine in which the local memories combine to form a global address space. Any node can specify any address in this global space. However, a node cannot read or write a non-local address directly. Remote addresses are accessed with special EARTH operations for remote access. A remote load is a *split-phase transaction* with two phases: *issuing the operation* and *using the value returned*. The second phase is performed in another thread, after the load has completed.

3.2 Threads

A thread is an atomically-scheduled sequence of instructions. When an EU executes a thread, it executes the instructions according to their sequential semantics. In other words, instructions within a thread are scheduled using an ordinary program counter. Notice that this does not preclude the use of semantically-correct out-of-order and parallel execution to increase the instruction issue rate within a thread. Both conditional and unconditional branches are only allowed to destinations within the same thread.

EARTH threads are *non-preemptive*. Once a thread begins execution, it remains active in the EU until it executes an EARTH operation to terminate the thread. If the CPU should stall (e.g., due to a cache miss), the thread will not be swapped out of the EU. There are no mechanisms to check that data accessed by an executing thread is actually valid or to suspend the thread if it isn't, except for normal register checks such as register scoreboarding. Therefore, data and control dependences must be checked and verified *before* a thread begins execution.

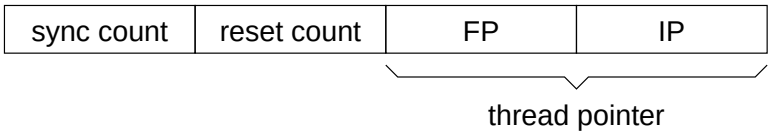

Figure 2: Synchronization slot format

This is done explicitly using *synchronization slots* and *synchronization signals*. A sync signal is used by the producer of a datum to tell the consumer that the data is ready. A sync slot is used to coordinate the incoming sync signals, so that a consumer knows when all required data is ready. Each sync signal is directed to a specific sync slot. Sync signals and slots are handled with explicit EARTH operations, and are made visible in Threaded-C.

3.2.1 Synchronization Slots

A sync slot contains three fields: a *reset count,* a *sync count,* and a *thread pointer*, as shown in Figure 2. The sync count indicates the number of sync signals that have to be received by the sync slot before the corresponding thread can be enabled. When a sync signal is received, the sync count is decremented. If the count reaches 0 the thread id specified by the thread pointer is placed in the ready queue and the sync count is set back to the value of the reset count. The thread pointer (or *thread id*) consists of a frame pointer (*FP*) and an instruction pointer (*IP*). The frame pointer specifies a particular function instance while the instruction pointer points to the first instruction of the thread.

For instance, consider the issue of a split-phase *read* operation in thread i. After issuing the read operation, thread i continues its execution without waiting for the read operation to finish. The operation that uses the returned value must be placed in a different thread, thread j. To ensure the correctness of the split-phase read, thread j is associated with a sync slot. This way, thread j will not be scheduled for execution before a sync signal is received signaling that the remote read has finished.

3.2.2 Thread Scheduling

The only restrictions imposed by the EARTH model on thread scheduling are the synchronization rules implemented by the sync slots. A thread can begin execution any time after the sync count in its corresponding sync slot reaches 0. The simplest scheduling policy is a FIFO scheduling in which newly enabled threads are placed in the end of the ready queue and the EU always read threads from the beginning of the ready queue. However, more elaborate policies are possible in implementations that allow random access to the ready queue [6]. For instance, threads can be prioritized to favor threads known to be on critical paths, or threads can be scheduled in LIFO order to benefit from register locality.

The atomicity and non-preemptiveness of threads does not prevent multiple threads from

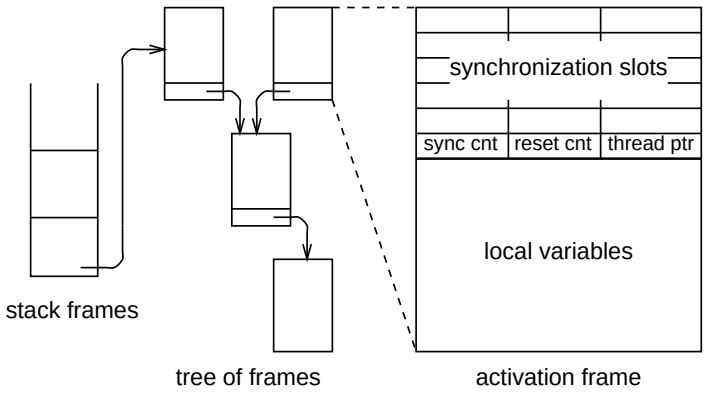

Figure 3: Stack and activation frames

running simultaneously on the same EARTH node. The EARTH model allows for the EU to maintain multiple independent active threads, whether their execution is interleaved or simultaneous. The model also allows for implementations in which several single-thread uniprocessors share the same memory, SU and ready queue. Therefore, to assure portability of code across all EARTH platforms, when writing Threaded-C code, a programmer shall make no unwarranted assumptions about concurrent thread execution.

3.3 Threaded Functions

A thread is always part of an enclosing *threaded function*. A threaded function is a piece of code containing local state (function parameters, local variables, and sync slots) and one or more threads of code. All threads in a function share the local variables. When a threaded function is invoked, a new frame is allocated in memory for this particular instance of the function. All threads access this frame through its *frame pointer*. Notice the similarity to the frame pointer mechanism in conventional block-structured languages. However, because multithreaded machines allow different parts of a program to be active at the same time, procedure frames cannot be placed on a simple stack as in conventional processors. EARTH is similar to other multithreaded and dataflow machines in that frames are dynamically allocated from a heap, as shown in Figure 3. Function frames can be allocated from arbitrary locations in memory. Although either the EU or the SU can perform such allocations, using the SU for memory allocation frees up the EU for thread execution.

Since all threads within a threaded function share the components of a single frame, a function instantiation has to run on only one processor. The caller can choose the processor in which a function instantiation will be executed. The EARTH system also provides an automatic processor assignment mechanism that tries to allocate a new function instantiation to the least loaded processor. Once a processor has been selected and the function begins to run, the function instance and its threads cannot be moved to another processor.

To exploit parallelism on a multi-processor EARTH, a programmer must partition a program

into function instances. If a function F_1 has to synchronize with another function F_2, then F_1 needs a pointer to the frame of F_2. Typically a function needs to synchronize with its caller to return results and signal termination. If each function has a pointer to the frame of its caller, a tree of function frame pointers is generated as shown in Figure 3. The frame in the lower center of the figure is of the caller function that originated the tree. Frame pointers point from children to parents as they link the call structure together. Note, however, that the synchronization slots permit the establishment of arbitrary synchronization patterns among a set of concurrently executing functions.

The EARTH programming model distinguishes between *threaded functions* and *sequential functions*. A *threaded function* contains one or more threads and can use all the Threaded-C language extensions. A *sequential function* is a standard function without any multithreading extensions. Threaded functions can call other threaded functions and sequential functions, but sequential functions can only call sequential functions. Sequential functions are implemented on a standard stack using a conventional call/return mechanism and therefore are not slowed by any multithreading overheads. The stack is only needed while such sequential code is executing, and is retained until the current sequential call is terminated. Since the runtime stack is a *sequential* implementation of potentially parallel function calls, *threaded functions* are not allowed within a sequential function because they may involve executing multiple functions in parallel. The left side of Figure 3 shows a stack that stores the sequential function frames.

4 Introduction to Threaded-C

This section introduces the basic features of Threaded-C and illustrates their use through real program examples. It begins with trivial examples (hello-world, Fibonacci) and ends with a small but real application (the N-queens enumeration problem). More advanced features of Threaded-C are covered in the next section.

4.1 The MAIN Function and Regular C Functions

Like sequential C programs, Threaded-C programs have a top-level main function. This function must be called `MAIN`, in capital letters, and be declared as a `THREADED` function. Its arguments are the same as for a sequential `main` function and have the same semantics. The last statement of the MAIN function must be the `RETURN()` primitive.

One of the simplest `MAIN` functions is shown in Figure 4. In line 5 we call the standard sequential printf function. Sequential C functions are declared and used in Threaded-C in exactly the same way that they are used in sequential C programs. There are no special restrictions on such functions.

As described in Section 3.3, the frame of a sequential C function is allocated from the runtime stack. Sequential C functions use the same calling mechanism as in normal sequential code, which means that they do not incur any multithreading overhead. Sequential C functions

```
1       #include    <stdio.h>
2
3       THREADED MAIN( int argc, char** argv )
4       {
5           printf("Hello World!\n");
6           RETURN();
7       }
```

Figure 4: A Simple Threaded-C Function

are *not* allowed to use any Threaded-C extensions, such as remote communication primitives, or call a threaded function.[4]

In the example above, regardless of how many nodes run the program, there will be only one "Hello World!" string printed. In contrast to other programming models, the MAIN function in Threaded-C is executed *only* on the first node of the system. However, it is possible to call threaded functions and let them execute on any node. In order to support this functionality, each node is loaded with an identical copy of the program code. The MAIN function begins running on the first node, designated node 0, and this function must explicitly distribute work to other nodes via function calls. On the other nodes, the runtime system simply waits for external requests and starts execution of individual threaded functions whenever requested.

4.2 Calling Functions on other Nodes

Since EARTH is a parallel machine, we might like to hear our favorite string from the remaining nodes as well. Threaded-C's `INVOKE` command allows us to call a threaded function on any node of the system. The arguments of `INVOKE` consist of the node id where we intend to run the function, the name of the function, and its arguments.

A threaded function can send/receive data to/from another node and generate parallel function calls, which a sequential C function cannot. The function frame of a threaded function is allocated dynamically from the heap, not from the runtime stack. A threaded function does not have a return value. Threaded functions must end with either `RETURN()` (as in the case of the MAIN function) or `END_FUNCTION()`.

If we print a string on another node, we might like to include some information about the node itself. The Threaded-C language defines two integer constants:

- `NUM_NODES` – the number of nodes that are executing the program
- `NODE_ID` – the number of the node ($0 \leq$ NODE_ID $< NUM_NODES$)

These are not *compile-time* constants, but *run-time* constants. They are initialized when the program is loaded, and can't be changed by an assignment statement. Thus, an EARTH

[4]There is one exception, namely the `POLL` command. This is discussed in a later section.

```
 1      #include    <stdio.h>
 2
 3      THREADED print_hello( void )
 4      {
 5          printf("From node %d: Hello World!\n", NODE_ID);
 6          END_FUNCTION();
 7      }
 8
 9      THREADED MAIN( int argc, char** argv )
10      {
11          int    i;
12
13          for (i = 0; i < NUM_NODES; ++i)
14              INVOKE(i, print_hello);
15          RETURN();
16      }
```

Figure 5: Hello World on All Nodes

executable can run without recompiling on two binary-compatible platforms with different numbers of processors. NUM_NODES is the same throughout the machine, while NODE_ID depends on the node. Thus, a function can determine on which node it is running.

With the explanations given above our first attempt might look like Figure 5. The output that we observe will depend on the number of nodes on which we are running the program. Even for a fixed number of nodes, the output might vary between runs. For instance, consider that we are observing the output of all the nodes on a single terminal. In this case the order in which the nodes will print may change from one run to the next.

What may call our attention is that some of the outputs may not be printed. The reason is that the thread doing the INVOKE is *not* suspended. Therefore, in lines 13 and 14 the MAIN function will request the execution of the print_hello function on all nodes and continue. The next action of the MAIN function, in line 15, is to return from execution. As a consequence the MAIN function might terminate before all the invoked print_hello functions were able to print. To fix this situation, we need a *synchronization mechanism* which allows us to impose a global order over thread execution.

4.3 Synchronization Between Functions

At the end of this section we present a thread-safe version of our parallel hello-world program. First we briefly discuss Threaded-C's synchronization mechanisms.

A *thread* in the context of multithreading is a sequence of instructions (within the scope of a threaded function) whose lexical order is determined at compile time. Threads run to completion, i.e., they are *non-preemptive*. Threads without dependences may run concurrently, and threads may generate and activate other threads through various mechanisms to be presented.

A function such as hello_world in Figure 5 consists of a single thread, which begins executing as soon as a frame for this function has been allocated by the runtime system. If more threads are desired, they must be delimited with special Threaded-C operators. To start a new thread, the preceding code is terminated with `END_THREAD()`, and then the new thread is started with the label `THREAD_`*i* (where *i* is a positive integer). These statements mark the *thread boundary*.

Thus, if a function has n threads, all threads but the first thread begin with a label THREAD_*i*; the first thread begins with the first executable statement of the function, and requires no explicit label. All threads but the last end with END_THREAD(); the last ends with END_FUNCTION() (or RETURN()). The first (unlabeled) thread is sometimes called the *initialization thread* because it automatically is eligible to begin execution once the function frame has been set up. No other threads in this function instance can start running until enabled by some synchronization event.

Synchronization in this context is the mechanism that allows us to impose some global order on thread execution. For instance, if we have two threads, THREAD_1 and THREAD_2, with a data dependence that demands that THREAD_1 executes before THREAD_2, then we must use synchronization to guarantee that THREAD_2's execution does not start before the execution of THREAD_1 ends.

In Threaded-C, synchronization is handled through *synchronization slots*. A sync slot has three fields: a sync count, a reset count and a thread identifier (the *i* in THREAD_*i*). The sync count changes dynamically, while the other two fields remain constant. The sync count holds the number of synchronizations that must be performed before the thread can be executed. Each incoming synchronization signal decrements the sync count by one. Once the sync count reaches zero, the corresponding thread is activated, and the sync value is reset to the reset value.

A threaded function has to declare the slots that it will use at the beginning of the function. This declaration must obey the following rules:

1. There is a single declaration, before any other local variable declarations.
2. The data type is `SLOT` and the name is `SYNC_SLOTS`.
3. The variable is declared as an array, even if only one sync slot is used.

The reset count, thread identifier, and the *initial* value of the sync count, are set using the `INIT_SYNC` command. In the simpler examples in this tutorial, the reset count is irrelevant, as the sync slot is only used once. However, real applications are likely to synchronize a thread repeatedly, in which case it is important that the reset count have the proper value. The initial value of the sync count ("init count") is usually the same as the reset count. However, it may be more or less, if the first occurrence of a thread has special synchronization requirements.

Now we are able to write a *synchronized* version of our hello-world program (Figure 6):

Line 3: The print_hello function takes a reference to a sync slot as argument. `SPTR` is a predefined Threaded-C data type for a pointer to a sync slot.

```
1       #include    <stdio.h>
2
3       THREADED print_hello( SPTR done )
4       {
5           printf("From node %d: Hello World!\n", NODE_ID);
6           RSYNC(done);
7           END_FUNCTION();
8       }
9
10      THREADED MAIN( int argc, char** argv )
11      {
12          SLOTS    SYNC_SLOTS[1];
13          int      i;
14
15          INIT_SYNC(0, NUM_NODES, NUM_NODES, 1);
16          for (i = 0; i < NUM_NODES; ++i)
17             INVOKE(i, print_hello, SLOT_ADR(0));
18          END_THREAD();    /* end of THREAD_0 */
19
20      THREAD_1:
21          RETURN();
22      }
```

Figure 6: Safe Hello World

Line 6: When print_hello finishes printing, it sends a sync signal to the sync slot.

Line 12: Only a single slot is needed, but SYNC_SLOTS is still declared an array.

Line 15: The first argument to INIT_SYNC indicates which slot is being initialized (slot 0). The next two arguments are the init count and reset count, respectively, Since we expect a sync signal from each node, we set the init count of slot 0 to NUM_NODES. The last argument identifies the thread (THREAD_1) to execute when a total of NUM_NODES syncs has been received.

Line 16: This loop is a simple example of a *forall* loop.

Line 21: `SLOT_ADR(`i`)` is a built-in macro generating a pointer to sync slot i (type SPTR). This is what gets assigned to `done` in Line 3.

Line 20: `THREAD_1` plays the role of a *barrier*. It won't run until all nodes have finished their printing.

4.3.1 Local and Remote Synchronization

Threaded-C provides two flavors of synchronization: local and remote. The preceding example shows remote synchronization. Remote synchronization is done with the RSYNC command, which takes a pointer to a sync slot (SPTR) as its argument. There is also a local form SYNC,

which can be used for synchronization *within* a function invocation. `SYNC(`i`)` (i an integer) sends a sync to local sync slot i.

Note that in this context, "remote" does *not* mean "on a different node," but merely "in a different frame." An SPTR is needed to refer to a sync slot in a different frame, even if that function invocation is running on the same processor. This convention is the same for all related EARTH operations: SYNC takes an int, which can only refer to a slot number within the particular function invocation, while RSYNC takes an SPTR, which can refer to any sync slot anywhere. (RSYNC can even refer to a local slot if it has been converted to an SPTR with the SLOT_ADR macro.)

4.3.2 Spawning Threads

Threaded-C provides an alternate mechanism for starting threads. A thread may be started directly using a spawn operation. A spawn command specifies a particular thread, which is placed in the ready queue immediately without having to be synchronized by a sync slot. A spawn operation is *not* equivalent to a transfer-of-control operation (such as a goto). If a spawn occurs in the middle of a thread, the thread performing the spawn continues to execute (since threads are non-preemptive), and the new thread could begin execution concurrently.

As with sync operations, there are local and remote spawn commands. The local SPAWN command takes an integer argument, which refers to a thread in the local function. The remote RSPAWN command is described in the appendix.

4.4 Load Balancing

Section 4.2 illustrated calling threaded functions on other nodes using the INVOKE command. This is highly effective for applications with predictable load distributions, for the programmer has explicit control over where each function runs. However, for many applications, especially irregular ones, the programmer may not have a good understanding of how to distribute data and the workload of the computations.

Therefore, Threaded-C has a "token" mechanism for calling functions on other node with automatic *load-balancing*. The syntax for `TOKEN()` is the same as for `INVOKE()` except that there is no argument to specify the node. Instead, it is up to the EARTH runtime system to decide where to run the function. The runtime system uses various heuristics based on the current state of the nodes to choose what it thinks is the best node to run the function.

In the code in the previous section, we could replace line 17 with the following line

```
TOKEN(print_hello, SLOT_ADR(0));
```

and we would still get the same number of "hello world" statements printed. However, the node numbers attached to the output line would be different, because the system is making its

own decisions about where to run each function. It is highly unlikely that each node would run exactly one function invocation. Indeed, many of the functions would probably run on node 0, for these functions are tiny, and some of them may finish before all the TOKEN commands have been executed.[5]

4.5 Transferring Data

The previous examples used pure synchronizations to control a distributed computation. Useful parallel computations need to exchange data too. Threaded-C provides mechanisms for sending data to and retrieving data from other functions. These mechanisms are coupled with the synchronization commands to make data-and-synchronization (data-sync) operators. The EARTH system guarantees the *atomicity* of these operators; in a combined data-sync operation, the data transfer is guaranteed to be complete before the synchronization is performed.

Data transfer operations need some place to put the data. As mentioned at the end of Section 3.1, EARTH is assumed to be a distributed memory machine with a global address space, in which processors cannot access addresses on remote processors *directly* (using load/store instructions), but may access them *indirectly* using EARTH operations. On some machines, the combined memory of all the processors may be too large to be addressed by regular pointers. Therefore, Threaded-C uses *global addresses* for EARTH operations between nodes.

A global address is like a regular pointer, but is distinguished by the keyword GLOBAL. For instance, `int *GLOBAL g` declares `g` to be a pointer to a global address containing an int. A global address is attached to a specific node, so it is possible to determine the node ID containing a specific address. Since this is implementation-dependent, special functions are provided for this (listed in the appendix). programmers should use these macros and not make assumptions about the internal representations of these pointers.[6]

One cannot dereference a global address directly. An operation such as `*g` is illegal. Therefore, we call this language construct a global *handle* instead of a global *pointer*, to prevent programmers from making unwarranted assumptions about what they can do with global addresses.[7] On the other hand, standard pointer arithmetic is allowed. One can convert explicitly between local pointers and global handles using EARTH operators, as shown in upcoming examples and in the appendix.

[5] Furthermore, some load balancers (depending on implementation) weigh the cost of sending something to another node against the benefit of sending work to an unloaded node.

[6] If the global address space in a particular machine is small enough to fit in a regular pointer, then the EARTH runtime system on that machine may use regular pointers for global addresses, in which case the GLOBAL keyword is ignored. However, programmers should always use the GLOBAL keyword to ensure their code is portable.

[7] In previous documents we have used the term *global pointer* to refer to this language construct.

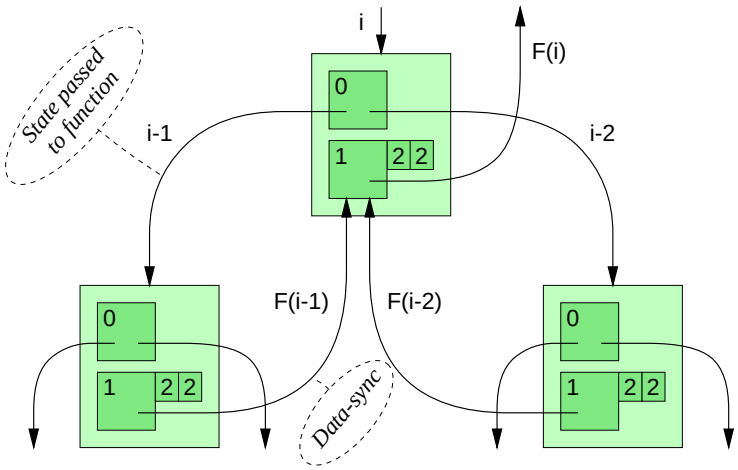

Figure 7: Fibonacci Code Structure

4.5.1 Simple Data Synchronization

The following example illustrates global handles and data-sync operations. We wish to compute Fibonacci numbers using a naïve recursive approach. If the recursive function fib(i) determines it is not a leaf, it calls fib($i-1$) and fib($i-2$), adding their results. Because distribution is hard to predict statically, we use tokens to take advantage of EARTH's load balancer. The recursive function must be split into two threads, like the MAIN function in our hello-world example, since the summing of the results is dependent on data sent back by the tokens.

The linkage of one function instance with its children is shown in Figure 7. The diagram shows each function instance as a light box, with two darker boxes representing the two threads. Box 0 in each function represents the initialization thread, which is not explicitly numbered; the other box is thread 1. Each thread (except initialization threads) shows the init count and sync count for the sync slot that controls that thread.[8]

The code for the Fibonacci recursive implementation is shown in Figure 8. The following explains the new commands introduced in this example:

Line 4: The parameters now include a global handle for receiving the result of the function (the n^{th} Fibonacci number).

Line 7: Local integers r1 and r2 receive the results of the recursive calls.

Lines 11–12: This function call is a leaf (no more recursion). Write the values 1 and 0 directly to r1 and r2, and spawn THREAD_1 directly.

Lines 14–15: This function call requires recursion. Call the function twice, with one result directed to r1 and the other directed to r2. The macro TO_GLOBAL converts the local

[8]It is assumed that each thread is controlled by a single sync slot. This is usually true in practice, but it *is* legal to have more than one sync slot control the same thread.

```
 1    #include    <stdio.h>
 2    #include    <stdlib.h>
 3
 4    THREADED fib( SPTR done, int n, int* GLOBAL result )
 5    {
 6        SLOT    SYNC_SLOTS[1];
 7        int     r1, r2;
 8
 9        INIT_SYNC(0, 2, 2, 1);
10        if (n < 2) {
11            r1 = 1; r2 = 0;
12            SPAWN(1);
13        } else {
14            TOKEN(fib, SLOT_ADR(0), n-1, TO_GLOBAL(&r1));
15            TOKEN(fib, SLOT_ADR(0), n-2, TO_GLOBAL(&r2));
16        }
17        END_THREAD();
18
19    THREAD_1:
20        DATA_RSYNC_L(r1 + r2, result, done);
21        END_FUNCTION();
22    }
23
24    THREADED MAIN(int argc, char** argv)
25    {
26        SLOT    SYNC_SLOTS[1];
27        int     N, res;
28
29        N = atoi(argv[1]);
30        INIT_SYNC(0, 1, 1, 1);
31        INVOKE(NODE_ID, fib, SLOT_ADR(0), N, TO_GLOBAL(&res));
32        END_THREAD();
33
34    THREAD_1:
35        printf("fib(%d) = %d\n", N, res);
36        RETURN();
37    }
```

Figure 8: Recursive Fibonacci

addresses &r1 and &r2 into global handles so that the two instances of fib() have places to send the results.

Line 20: This is run when the children have sent their results back to r1 and r2. Add the sub-results together and use a remote data-sync variant to send the sum to the caller. The _L suffix indicates the type of the first argument, the value to be sent (long).

Line 24: The MAIN function is similar to the MAIN function in the hello-world example, except that it does some command-line processing and only requires one sync.

For each type of data-sync operation, there is a corresponding *data-spawn* operation. These use the spawning mechanism (see Section 4.3.2) rather than sync slots, but are otherwise equivalent, and guarantee atomicity (completion of the data transfer before the spawn occurs).

4.5.2 Fetching Data with Synchronization

The Fibonacci example uses data-syncs for data transfers. This works fine for programs in which the *producer* of data always knows where to send it, i.e., knows the identity of the *consumer*. However, there are situations where the situation is reversed: the producer doesn't know where to send data at the time it is produced, and it is up to the consumer to request the data. Since a remote fetch can be a long-latency operation, we want to switch to another thread during this time to hide this latency.

Therefore, such a fetch should be a *split-phase transaction*. The request for the data and the code that uses this data should be in different threads. For each data-sync operation in Threaded-C, there is a corresponding *get-sync* operation. For instance, the DATA_SYNC_L instruction in line 20 sends the first argument (a long) to the global handle `result` (which points to a long) and syncs the sync slot with global handle `done`. An example of a corresponding get-sync function would be:

```
GET_SYNC_L(remote, TO_GLOBAL(&local), SLOT_ADR(0));
```

(assuming `remote` is a global handle pointing to a long, and `local` is a local long). This instruction would send a request to the node owning the address pointed to by `remote`, requesting that the contents of that location be sent to `local`, and that the local sync slot 0 be synchronized. The request is processed by the synchronization unit on the remote node, *without the intervention of the user code*. The remote SU sends a message back to the local SU, which updates the local address and syncs the slot. Presumably, the thread guarded by slot 0 will use the value in `local`.

The next section shows a program example that uses a get-sync operation. As with data-syncs, there are several variants of this class of operation. The GET_RSYNC_* commands specify a sync slot using a global handle rather than a slot number. Also, there are get-spawn equivalents of all the get-sync operations (as shown in the appendix).

4.5.3 Block Moves

The data-sync and get-sync primitives only allow a single datum to be moved. Threaded-C allows contiguous blocks of arbitrary size to be moved. As with the get-sync operations, each operation takes two data addresses and a reference to a sync slot. An additional argument specifies the size of the block. Block moves are similar to the memcpy() function in C: addresses are generic (untyped), and the size argument specifies the size in bytes.[9]

[9] A block move involving only local blocks (both source and destination) would be equivalent to a memcpy() followed by a local SYNC. However, it is better to use block moves in such situations. If the machine has a

A block move operation will copy the block starting at the specified source address directly to the destination address. Observe that block moves are call-by-reference, and the runtime system will not make an intermediate copy of the data. *Therefore it is the programmer's responsibility to guarantee that the data at the source will remain unchanged until the specified synchronization slot is synchronized by the operation.* The BLKMOV_SYNC operation is used when the synchronization slot is local to the function requesting the block move. When the synchronization slot is in a different function, the command to use is BLKMOV_RSYNC.

We illustrate the use of block moves with a recursive program that is similar to the previous Fibonacci example, but requires block move operations. The N-queens code counts the number of ways in which N queens can be placed on an $N \times N$ chessboard so that no queen can attack any other queen (under normal chess rules). The code uses a recursive divide-and-conquer strategy.

A queen is considered to be in a valid position if it cannot attack any other queen in the chessboard. A partial solution is represented by a 1-dimensional array. Each element of this array corresponds to a row of the chess board and contains the column in which a queen is positioned in that row. Initially a queen is placed in the position $(0, 0)$, creating a solution array with a 0 in the first position and non-valid entries in all other positions.

A search procedure is called with a row number r, a column number c, and a partial solution array, in which rows 0 through $r - 1$ have been filled with queens in valid position. The procedure returns the number of complete solutions in which rows 0 through $r - 1$ match the initial configuration, and in which the queen in row r is somewhere between columns c and $N - 1$ (inclusive). It works by trying the queen in all positions from column c to the right side of the board. If it finds a valid position, it splits the search into two sub-searches, as shown in Figure 9:

- Add the new queen to the chessboard and start searching the next row.
- Keep trying positions to the right of the current position.

The top-level function calls the search procedure with an empty board ($r = c = 0$).

The control structure is similar to our Fibonacci example, suggesting that a code structure similar to Figure 7 is appropriate. But there is one problem. The state of the computation is represented by a one-dimensional array. In a sequential implementation, this array can be continually updated in place, but not in a parallel implementation, where the parallel function instances would interfere with each other. Furthermore, function instances on different nodes can't share the same array, since EARTH assumes a distributed memory machine. Instead, we need to replicate the state whenever we start a new function. However, since arrays can't be passed "by value" in C, the partial solution can't be directly passed to the next level in the parameter list.

separate synchronization unit, this separate unit can perform the block move while the CPU does other useful work. If there is no separate unit to perform a block move, the compiler will convert the block move command into a memcpy() anyway.

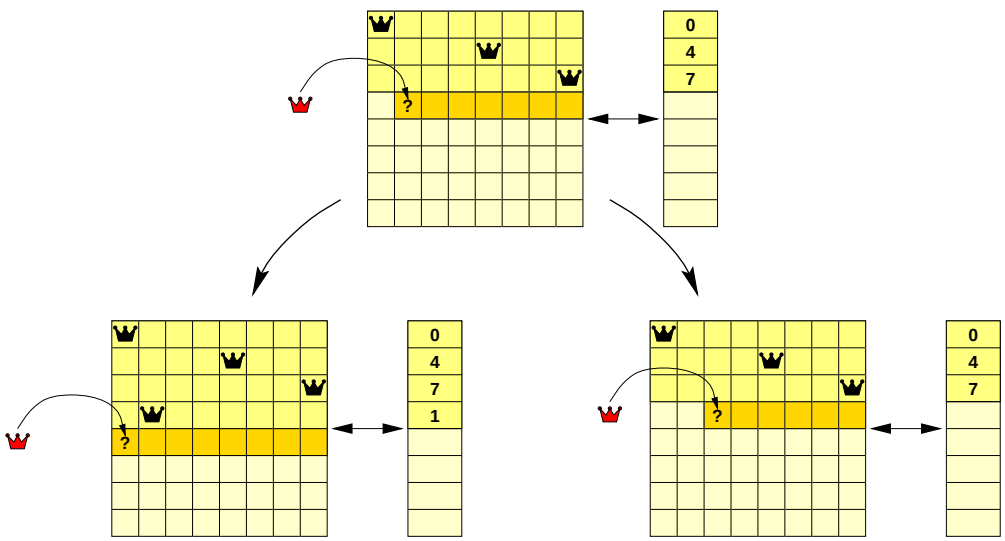

Figure 9: Recursion in N-Queens

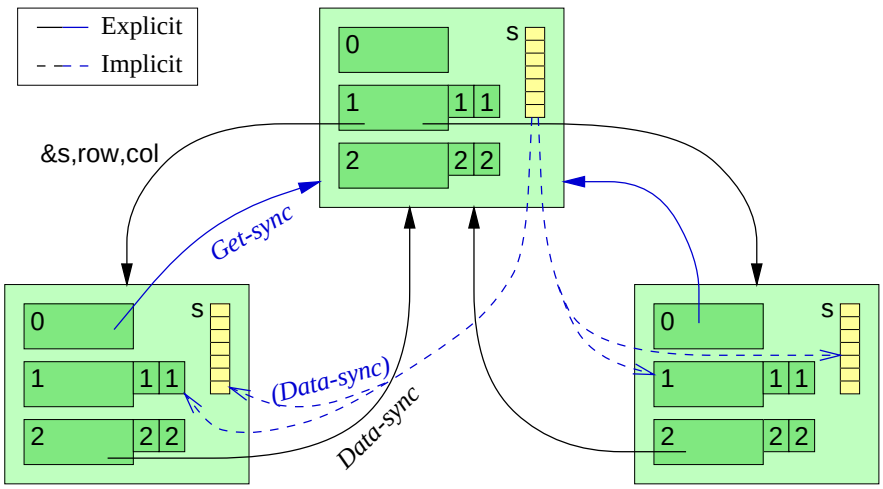

Figure 10: Data Transfers and Synchronization in the N-Queens Solution

The solution is for a function to fetch a copy of its parent's state using a split-phase transaction. We need to split the initialization thread in Figure 7, which performs the recursion, into two threads: one to initiate the fetch and one to perform the recursion once the fetch has completed. The caller passes a pointer to its copy of the partial solution. The callee then *fetches* the block with a block-move. In our implementation, the initialization thread issues the block move request, and THREAD_1 receives the data. This thread then invokes the new search procedures (using TOKEN commands), and the results must synchronize THREAD_2. The structure is illustrated in Figure 10, where dashed lines indicate data transfers that are performed automatically by the EARTH runtime system (as a consequence of the EARTH operations used in the code).

Figure 11 lists the recursive search routine. The program assumes a maximum of 24 queens.

```
1   THREADED queens(int *GLOBAL result,   /* Where to send result */
2                   SPTR done,            /* Slot to sync when sent */
3                   int n,                /* Size of board */
4                   int row,              /* Current row (fixed above) */
5                   int start_col,        /* Check from here to end */
6                   int *GLOBAL previous) /* Caller's board state */
7   {
8           SLOT    SYNC_SLOTS[2];
9           int     col,                  /* Column index */
10                  own_board[24],        /* Local copy of board */
11                  sols_this_col,        /* # solns with this position */
12                  sols_other_cols;      /* # solutions to right side */
13
14    INIT_SYNC(0, 1, 1, 1);
15    INIT_SYNC(1, 2, 2, 2);
16    BLKMOV_SYNC(previous, TO_GLOBAL(own_board), row*sizeof(int), 0);
17    sols_this_col = sols_other_cols = 0;     /* Preset sub-solutions */
18    END_THREAD();
19
20  THREAD_1:                /* Activated when block move is complete */
21    for (col = start_col; col < n; col++) {       /* Here to right end */
22      if (safe(own_board, row, col)) {
23        own_board[row] = col;
24        if (row+1 == n) {       /* Board full?  This is a solution! */
25          sols_this_col = 1;    /* Set the partial result */
26          SYNC(1);              /* Sync directly instead of TOKEN */
27        } else {                /* How many solns with this partial? */
28          TOKEN(queens, TO_GLOBAL(&sols_this_col), SLOT_ADR(1),
29                n, row+1, 0, TO_GLOBAL(own_board));
30        }
31        if (col+1 == n) {       /* Last column?  No more solutions */
32          SYNC(1);              /* Sync directly instead of TOKEN */
33        } else {                /* How many solutions to the right? */
34          TOKEN(queens, TO_GLOBAL(&sols_other_cols), SLOT_ADR(1),
35                n, row, col+1, TO_GLOBAL(own_board));
36        }
37        break;
38      }
39    }
40    if (col == n)       /* No safe position found?  Then return 0 */
41      SPAWN(2);         /* Sub-solns already 0; start final thread now */
42    END_THREAD();
43
44  THREAD_2:             /* Activated when both sub-solutions computed */
45    DATA_RSYNC_L(sols_this_col + sols_other_cols, result, done);
46    END_FUNCTION();
47  }
```

Figure 11: Recursive N-Queens Code

```
 1  int safe (int board[], int row, int col)
 2  {
 3    int rowchk, colchk;
 4
 5    for (rowchk = 0; rowchk < row; rowchk++) {
 6      colchk = board[rowchk];
 7      if ((col == colchk) || (row - rowchk == col - colchk) ||
 8          (row - rowchk == colchk - col))          /* Conflict? */
 9        return 0;
10    }
11    return 1;
12  }
13
14  THREADED MAIN (int argc, char *argv[])
15  {
16    SLOT SYNC_SLOTS[1];
17    int n, result, place[24];
18
19    if(argc != 2)
20      exit(1);
21    n = atoi(argv[1]);
22    INIT_SYNC(0, 1, 1, 1);
23    INVOKE(0, queens, TO_GLOBAL(&result), SLOT_ADR(0),
24           n, 0, 0, TO_GLOBAL(place));
25    END_THREAD();
26
27  THREAD_1:
28    printf("Number of solutions for %d queens: %d\n", n, result);
29    RETURN();
30  }
```

Figure 12: N-Queens Checking Function and MAIN Routine

A more general solution would allow arbitrary board sizes and use malloc() instead of frame space for the local copy of the state. The threaded code uses the sequential routine safe() to verify the validity of a partial solution. The code for this function, as well as the MAIN routine which makes the initial call to queens(), is listed in Figure 12.

Line 16 of Figure 11 asks for a copy of the remote board state to be copied into own_board[]. Since the second arg (destination) is local, this block move is similar to the get-sync commands discussed in the previous section (which copy remote data to local memory in a split-phase transaction), except that more than one datum is moved. Since we are syncing a local sync slot (0), we use BLKMOV_SYNC, whose last argument specifies a local slot number. Block moves can also be used like the data-syncs introduced in Section 4.5.1, sending local data to a remote destination and synchronizing a slot at the far end. For these, we need to use BLKMOV_RSYNC, whose last argument, like DATA_RSYNC, is a global handle of a slot (SPTR) rather than a local slot number (int).[10]

[10] Actually, block moves are more general. Any arbitrary combination of local and remote addresses is possible,

```
1       float *f_table;       /* Global-scope variable */
2
3       THREADED copy_table( float *GLOBAL table, SPTR done, int elems )
4       {
5           f_table = (float *) malloc(elems*sizeof(float));
6           BLKMOV_RSYNC(table, TO_GLOBAL(f_table),
7                        elems*sizeof(float), done);
8           END_FUNCTION();
9       }
```

Figure 13: Function to Replicate Table

5 Advanced Features of Threaded-C

In this section we discuss more advanced features of Threaded-C, including global-scope variables and polling.

5.1 Variables in Global Scope and Memory Allocation

As previously mentioned, the executable code is replicated on all nodes, so that any function can run on any node. The same is true for *global-scope* variables, which are variables declared outside of the scope of a function (sequential or threaded). Such variables are replicated over nodes without explicit coherence mechanisms. Therefore if we declare a variable in global scope, the address of this variable will be the same in all nodes. However, each node will have its own independent copy of the variable.

Global-scope variables can be useful in some situations. For instance, suppose a program generates many tokens, each of which may read data from a read-only array (for instance, a lookup table of pre-computed values). Each function could fetch individual data it needs from a common source using get-sync operations, or could load the entire table using a block move (as was done by N-queens). However, the former might introduce too much latency, and the latter would be wastefully redundant if each node were to run many instances of the same function.

Instead, it would be better to make *one* copy of the table on each node, and put each copy in a place where any function knows where to find it. Figure 13 shows a simple function to copy the data. Each node has a pointer called f_table. Line 5 allocates an array big enough to hold the specified number of floats and makes f_table point to it.[11] Lines 6–7 copy the data from the source (at the given global handle) to the local block, and syncs the *remote* sync slot passed to the function. Note that the second argument points to the array that was just malloc'ed,

since all address arguments must be global handles. For instance, code on node 0 could copy an array from an address in node 1 to another in node 2, and sync a slot in node 3. The runtime system determines dynamically what type of transfer is being requested, since global handles are used for both source and destination, and the runtime system guarantees that the data will be completely copied before the sync slot is touched. However, such uses are rare, and the vast majority of block moves are of the data-sync and get-sync flavors.

[11] Code to check if malloc returns NULL has been removed for clarity.

```
 1      THREADED allocate( SPTR done, int** here, int size )
 2      {
 3          int     i;
 4
 5          *here = (int*) malloc(size*sizeof(int));
 6          for (i = 0; i < size; ++i)
 7              (*here)[i] = NODE_ID;
 8          RSYNC(done);
 9          END_FUNCTION();
10      }
11
12      THREADED deallocate( SPTR done, int** here )
13      {
14          free(*here);
15          RSYNC(done);
16          END_FUNCTION();
17      }
```

Figure 14: Allocate and Deallocate with Global-Scope Variables

not the global-scope variable itself. (An argument such as `TO_GLOBAL(&f_table)` *would* point to the global-scope variable, and would certainly be an error!)

The MAIN routine could replicate the table on all the nodes by using INVOKE (*not* TOKEN) to invoke copy_table on each node, similar to our original hello-world example (Figure 6). The MAIN routine should use the syncs returned by copy_table to synchronize a barrier, so that MAIN doesn't invoke any tokens that will use the table data until all copy operations have completed. Once this is done, anyone can read the table, e.g., `d = f_table[i]`.

There's a small *subtlety* in the operation of this routine. The local copies of the pointer f_table have the same address on each node. However, the addresses to which those pointers *refer* (the base of each data table), may be *different* because we have a distributed runtime system, and the heap may be in different states on different nodes.

A more detailed example follows. In this example we want to allocate and deallocate blocks of data from local heaps and assign them to global-scope variables. In this example we allocate an array of integers in each node of the machine. We write a THREADED wrapper for the standard malloc function (see Figure 14. To simplify our allocation task, we define a pointer in global scope to hold the address of the first position of the array.[12]

In this code, observe that in Line 1 the arguments are passed a reference to a sync slot, the address of the pointer to which we want to attach the array, and the size of the array to be allocated. In Line 7 we initialize the array with the NODE_ID. The counterpart of the allocate() function is deallocate().

[12]Although not elegant, this is an efficient construct. Without the global-scope variable a more extensive allocation bookkeeping would be necessary. Future versions of Threaded-C will provide more transparent addressing/naming solutions.

```
 1      int*    ARRAY;
 2
 3      THREADED MAIN( int argc, char** argv )
 4      {
 5          SLOT    SYNC_SLOTS[2];
 6          int     i;
 7
 8          INIT_SYNC(0, NUM_NODES, NUM_NODES, 1);
 9          for (i = 0; i < NUM_NODES; ++i)
10              INVOKE(i, allocate, SLOT_ADR(0), &ARRAY, (i+1)*1000);
11          END_THREAD();
12
13      THREAD_1:
14          INIT_SYNC(1, NUM_NODES, NUM_NODES, 2);
15          for (i = 0; i < NUM_NODES; ++i)
16              INVOKE(i, deallocate, SLOT_ADR(1), &ARRAY);
17          END_THREAD();
18
19      THREAD_2:
20          RETURN();
21      }
```

Figure 15: MAIN Routine Calling Allocate and Deallocate

Following these functions we present the code of a MAIN function that uses them (Figure 15). In Line 1 we define a pointer variable in global scope. Note that in Line 12 we pass the address of ARRAY (`&ARRAY`) as an argument to allocate(). Since the addresses of global-scope variables are the same over the nodes this "hack" works.

We conclude this section with an alternative approach in which we do not use variables in global scope (Figure 16). Note that efree() has only a global handle argument. In Line 30 the function figures out (via `OWNER_OF`) in which address space the free operation must be executed. Figure 17 shows code which uses ealloc() and efree().

In Line 8 we allocate an array of global handles in node 0. Its entries refer to arrays which we dynamically allocate in other nodes. This kind of *bookkeeping* is inevitable in the current Threaded-C.

The examples in this section showed how pointers in global scope can be useful in certain situations. However, global-scope variables, like global variables in conventional languages, should be "considered harmful" and used only sparingly.

5.2 Polling

Section 2.1 described how implementations of EARTH on nodes without a second processor or hardware to support communication have to perform both the EU and SU functions in the same processor. A problem arises when a node issues a request for data on a remote processor

```
 1   typedef    void*GLOBAL        GlobalHandle;
 2   typedef    void*GLOBAL*GLOBAL    GlobalHandle2;
 3
 4   THREADED ealloc(SPTR done, GlobalHandle2 hp, size_t size)
 5   {
 6       void*    ptr;
 7
 8       ptr = malloc(size);
 9       printf("node %d: %d x sizeof(char) allocated at 0x%x.\n",
10              NODE_ID, size, ptr);
11       DATA_RSYNC_G(TO_GLOBAL(ptr), hp, done);
12       END_FUNCTION();
13   }
14
15   THREADED free_wrapper(SPTR done, GlobalHandle there)
16   {
17       printf("node %d: freeing 0x%x...\n", NODE_ID, TO_LOCAL(there));
18       free(TO_LOCAL(there));
19       RSYNC(done);
20       END_FUNCTION();
21   }
22
23   THREADED efree(GlobalHandle h)
24   {
25       SLOTS    SYNC_SLOTS[1];
26       int    node;
27       void*    ptr;
28
29       INIT_SYNC(0, 1, 1, 1);
30       node = OWNER_OF(h);
31       INVOKE(node, free_wrapper, SLOT_ADR(0), h);
32       END_THREAD();
33
34   THREAD_1:
35       RETURN();
36   }
```

Figure 16: Allocate and Deallocate without Global-Scope Variables

(get-sync or block move) and the remote processor is executing a thread. For efficiency reasons, the incoming message should not necessarily interrupt the running thread. Polling is much more efficient than interrupts, so the processor should finish the thread, and *then* poll the network for incoming messages.

Therefore, on single-processor-node implementations without special hardware support, the EU will poll the network every time it switches to a new thread. However, if the thread is long, the processor which issued the request may have to wait a long time for the data it requested, which could cause it to stall if that data is on a critical path. Some Threaded-C compilers may try to insert polls into the code that it generates if it determines that a thread may be long.

```
 1  THREADED MAIN(int argc, char** argv)
 2  {
 3      SLOT        SYNC_SLOTS[1];
 4      int        i;
 5      GlobalHandle*    table;
 6
 7      INIT_SYNC(0, NUM_NODES, NUM_NODES, 1);
 8      table = (GlobalHandle*) malloc(NUM_NODES*sizeof(GlobalHandle));
 9      for (i = 0; i < NUM_NODES; ++i)
10          INVOKE(i, ealloc, SLOT_ADR(0),
11                 TO_GLOBAL(table+i), (i+1)*1000);
12  END_THREAD();
13
14  THREAD_1:
15      for (i = 0; i < NUM_NODES; ++i)
16          CALL(efree, table[i]);
17      free(table);
18      RETURN();
19  }
```

Figure 17: MAIN Function Using Global-Free Allocate and Deallocate

However, the compiler may not always guess correctly, and is likely to be conservative in order not to impede the normal running threads.

Therefore, Threaded-C provides a POLL() operation, which takes no arguments. The POLL command has no effect on the semantics of the Threaded-C code. It is merely a hint to the compiler (or pragma) suggesting that here would be a good spot to poll the network. Some compilers may choose to make their own decisions about where to put polls. On machines which can automatically handle incoming messages without EU intervention (e.g., the MANNA), the POLL command is simply ignored. But programmers may wish to insert them into long threads for portability.

POLL commands can be used inside of both threaded and sequential functions. They are the *only* Threaded-C command which can be used inside a sequential function.

6 Examples of Threaded-C Programming

In this section, we illustrate the techniques covered in the previous two sections with some larger, more complex code examples. The first example shows how to implement mutual exclusion in Threaded-C. The second example shows one way to implement FFT.

6.1 Mutual Exclusion in Threaded-C

Currently, Threaded-C does not provide direct support for the implementation of mutual exclusion based on atomic operations. However there is a straightforward implementation of mutual

exclusion using the currently available primitives in Threaded-C. This implementation assumes that the function containing the mutual exclusion computation consists of a single thread, and that the shared variables used in the mutual exclusive computation are not modified by any other function.

The mutual exclusion code presented in this section also assumes that the Threaded-C implementation adheres to the following assumptions:

1. Threads are non-preemptive.
2. Only a single thread can run on a node at a time.

Observe that while condition 1 is inherent to the EARTH model, condition 2 might not be valid in future implementations (in SMP clusters for example). However, it is true for all current implementations of EARTH systems.

We illustrate the use of mutual exclusion assuming a "black board" multiprocessing organization in which each individual node performs its computation and every once in a while reports its intermediate results to a shared data structure. The reporting node receives an answer with the best value reported so far. If a node just reported the best value, it will receive its own value back. The mutual exclusion is necessary to ensure that the updating of the shared value is an atomic operation.

We choose to use static variables to store the shared memory values that must persist throughout the computation. The drawback of this implementation is that the address of this variable needs to be passed as an argument to all functions that interact with the corresponding mutual exclusion region. Alternative implementations could use dynamic allocation in the heap of the node that is in charge of the mutual exclusive computation or variables in global scope.

The mutual exclusion is based in the fact that a single node, called the MUTEX_NODE in our code, is chosen to process all the mutual exclusive computation. Because we assume that at most one thread can be active in each node, mutual exclusion is guaranteed. All the nodes that need to interact through mutual exclusion must invoke the corresponding thread in the mutex node.

The following present the code that implements a mutex synchronization mechanism for a black board architecture using standard Threaded-C primitives. Some of the important constructions to notice in this code are:

Line 1: Specifies that the node in charge of the mutual exclusion region processing is node 4.

Lines 14-15: Invokes a thread to initialize the shared value in the node that is the book keeper of the shared value (mutex node). We have to pass to the function the address of our pointer that will store the global address of the shared variable. The initializing function will write in our pointer. Therefore we can pass this global address to all the functions that use the mutual exclusive region.

```
 1  #define  MUTEX_NODE  4
 2
 3  THREADED MAIN(int argc, char** argv)
 4  {
 5      SLOT       SYNC_SLOTS[3];
 6      int        i;
 7      int        final;
 8      int GLOBAL *shared_value;
 9
10      INIT_SYNC(0,1,1,1);
11      INIT_SYNC(1, NUM_NODES, NUM_NODES, 2);
12      INIT_SYNC(2,1,1,3);
13
14      INVOKE(MUTEX_NODE, initialize_shared_value, 0,
15             TO_GLOBAL(&shared_value), SLOT_ADR(0));
16      END_THREAD();
17
18  THREAD_1:
19
20      for(i = 0; i < NUM_NODES; ++i)
21         INVOKE(i, produce_value, MUTEX_NODE, shared_value, SLOT_ADR(1));
22      END_THREAD();
23
24  THREAD_2:
25
26      INVOKE(MUTEX_NODE, print_shared_value, shared_value, SLOT_ADR(2));
27      END_THREAD();
28
29  THREAD_3:
30
31      RETURN();
32  }
```

Figure 18: Mutual Exclusion – MAIN Routine

Lines 18-22: Thread 1 invokes the function that produces new values in each node in the system, this function receives the number of the mutex node, and a sync slot to be synchronized at the end of the computation. Remember that shared_value contains now the the global address of the shared value.

Lines 24-26: Thread 2 is spawned when all the nodes have reported (through synchronization of slot 1) that they have finished performing their computations. We now ask the mutex node to print the final shared value.

Lines 29-31: Thread 3 (spawned by the synchronization of slot 2) is only necessary to assure that the computation does not terminate before the final value is printed.

The function produce_value (Figure 19) illustrates how a new local value is reported to the node that keeps the shared value. We are assuming that we also want a reply with the current

```
32  THREADED produce_value(int mutex_node, GLOBAL int *shared_value,
33                         SPTR finished)
34  {
35      int local_value;
36
37      SLOT   SYNC_SLOTS[1];
38      INIT_SYNC(0,1,1,1);
39
40      local_value = NODE_ID
41
42      INVOKE(mutex_node, mutex_update, local_value, shared_value,
43             TO_GLOBAL(&local_value), SLOT_ADR(0));
44      END_THREAD();
45
46  THREAD_1:
47
48      RSYNC(finished);
49      END_FUNCTION();
50  }
```

Figure 19: Mutual Exclusion – Produce Value

```
51  THREADED mutex_update(int new_value, int GLOBAL *shared_value,
52                        int GLOBAL *reply_address, SPTR reply_slot)
53  {
54     printf("Value %d was reported. ",new_value);
55
56     if(*shared_value < new_value)
57       *shared_value = new_value;
58
59     printf("Mutex node replied with %d.\n",
60            *(int *)TO_LOCAL(shared_value));
61     DATA_RSYNC_L(*shared_value, reply_address, reply_slot);
62     END_FUNCTION();
63  }
```

Figure 20: Mutual Exclusion – Update Value

shared value before we proceed with the computation.

When thread 1 (line 46) is spawned, the variable local_value has the most recent shared value reported by the mutex node. We could now proceed with local computation using this new value, though in this example, we simply report to the node that invoked this function.

The mutex_update() function (Figure 20) contains the computation to be done in the mutually exclusive region. In this simple example we just maintain the maximum value reported by all the nodes in the shared variable. We report the new maximum value to the node that invoked this function.

```
64  THREADED initialize_shared_value(int initial_value,
65                                   int GLOBAL **shared_value,
66                                   SPTR initialized)
67  {
68    static my_shared;
69
70    my_shared = initial_value;
71    DATA_RSYNC_G(TO_GLOBAL(&my_shared), shared_value, initialized);
72    END_FUNCTION();
73  }
74
75  THREADED print_shared_value(int GLOBAL *shared_value, SPTR printed)
76  {
77    printf("Final maximum value: %d.\n",*(int *)TO_LOCAL(shared_value));
78    RSYNC(printed);
79    END_FUNCTION();
80  }
```

Figure 21: Mutual Exclusion – Initialize and Print

The initialize_shared_value() function (Figure 21) must allocate all the static variables necessary for the computation in the mutual exclusion region. The address of the allocated variable is returned to the main function using the DATA_RSYNC_G primitive. These addresses will be passed to all the nodes that interact through the mutual exclusion region.

6.2 Example: One-Dimensional Fast Fourier Transform

We conclude this section with a more challenging example: the parallelization of a 1D FFT. The program below is a first, straightforward approach to this problem. This implementation of the 1D FFT in Threaded-C works fine if the problem is not too small and the number of processors is not too large. However, it is a naïve solution that neglects certain constraints of a real architecture.

Algorithmic Background: Consider a sequence

$$f = \{f(0), f(1), \ldots, f(2^N - 1)\}$$

of 2^N complex numbers. Its Fourier transform

$$F = \{F(0), F(1), \ldots, F(2^N - 1)\}$$

is given by

$$F(j) = \frac{1}{2^N} \sum_{k=0}^{2^N-1} f(k) W_N^{-jk}$$

where

$$W_N = \exp(2\pi\imath/2^N).$$

The even and odd parts of f are given by

$$f_0 = \{f(0), f(2), \ldots, f(2^N - 2)\}$$

and

$$f_1 = \{f(1), f(3), \ldots, f(2^N - 1)\}$$

respectively. Let F_0 and F_1 denote their Fourier transform. It can be easily shown that for $j = 0, ..., 2^{N-1} - 1$:

$$F(j) = \frac{1}{2}\left[F_0(j) + W_N^{-j} F_1(j)\right] \quad (1)$$

$$F(j + 2^{N-1}) = \frac{1}{2}\left[F_0(j) - W_N^{-j} F_1(j)\right] . \quad (2)$$

The formulas (1) and (2) show that the Fourier transform of a vector of length 2^N can be obtained from the Fourier transform of its even and odd parts, which are of half length. In the next section we discuss a Threaded-C implementation of the core routine of the corresponding *recursion*. It illustrates the use of BLKMOV_SYNC and shows a non-artificial application of the TOKEN mechanism.

The Core Routine in Threaded-C: The MAIN function does nothing but call the function fft_token() on the node where the initial data vector resides.

Line 1: The arguments of fft_token() consist of a global handle to a sync slot, a global handle to the array to be transformed, the log of its length, and finally, a sign (± 1) indicating whether we do a forward or backward Fourier transform.

Line 14: We check whether we already reached the bottom of the recursion.

Line 20: We fetch the array to a local address and overlap the retrieval with the computation of the necessary complex roots of unity.

Line 29: c_exp() computes the complex exponential function.

Line 32: c_scale() scale a complex number with a real number.

Line 46: We shuffle the even part of the array into the lower half of data_even and the odd part goes into the upper half.

Lines 48–49: This is the proper recursion part: we submit two tokens to do the transform of the even and odd parts. The communication pattern of the FFT is very regular. Therefore, one could try to do the load balancing by hand and use INVOKE instead of TOKEN.

Line 56: c_copy() copies a complex number.

Line 57: c_mult() computes the product of two complex numbers.

```
1     THREADED fft_token(SPTR done, Complex *GLOBAL data, unsigned int N, int isign)
2     {
3       SLOT    SYNC_SLOTS[5];
4
5       unsigned long   i, length, half_length;
6       Complex         *data_even, *data_odd, *w;
7       Complex         phase, z1, z2;
8
9       z1.re = z1.im = z2.re = z2.im = 0.0;
10
11      length = 1 << N;
12      half_length = length >> 1;
13
14      if (N > 1) {
15        INIT_SYNC(0, 1, 1, 1);
16
17        data_even = (Complex*) malloc(length*sizeof(Complex));
18        data_odd = data_even + half_length;
19
20        BLKMOV_SYNC(data, TO_GLOBAL(data_even), length*sizeof(Complex), 0);
21
22        phase.re = 0.0;
23        phase.im = M_PI * ((double) isign) / (double) half_length;
24
25        w = (Complex*) malloc(half_length*sizeof(Complex));
26
27        w[0].re = 1.0;
28        w[0].im = 0.0;
29        c_exp(w+1, phase);
30
31        for (i = 2; i < half_length; ++i) {
32          c_scale(&z1, (double) i, phase);
33          c_exp(w+i, z1);
34        }
35     } else {
36        INIT_SYNC(3, 1, 1, 3);
37
38        data_even = (Complex*) malloc(2*sizeof(Complex));
39
40        BLKMOV_SYNC(data, TO_GLOBAL(data_even), 2*sizeof(Complex), 3);
41      }
42      END_THREAD();   /* end of THREAD_0 */
43    THREAD_1:
44      INIT_SYNC(1, 2, 2, 2);
45
46      shuffle(data_even, N);
47
48      TOKEN(fft_token, SLOT_ADR(1), TO_GLOBAL(data_even), N-1, isign);
49      TOKEN(fft_token, SLOT_ADR(1), TO_GLOBAL(data_odd), N-1, isign);
50
51      END_THREAD();    /* end of THREAD_1 */
```

```
52    THREAD_2:
53      INIT_SYNC(2, 1, 1, 4);
54
55      for (i = 0; i < half_length; ++i) {
56        c_copy(&z1, data_even[i]);
57        c_mult(&z2, w[i], data_odd[i]);
58        data_even[i].re = z1.re + z2.re;
59        data_even[i].im = z1.im + z2.im;
60        data_odd[i].re = z1.re - z2.re;
61        data_odd[i].im = z1.im - z2.im;
62      }
63
64      BLKMOV_SYNC(TO_GLOBAL(data_even), data, length*sizeof(Complex), 2);
65      free(w);
66      END_THREAD();   /* end of THREAD_2 */
67    THREAD_3:
68      INIT_SYNC(4, 1, 1, 4);
69
70      c_copy(&z1, data_even[0]);
71      c_copy(&z2, data_even[1]);
72
73      data_even[0].re = z1.re + z2.re;
74      data_even[0].im = z1.im + z2.im;
75      data_even[1].re = z1.re - z2.re;
76      data_even[1].im = z1.im - z2.im;
77
78      BLKMOV_SYNC(TO_GLOBAL(data_even), data, 2*sizeof(Complex), 4);
79      END_THREAD();   /* end of THREAD_3 */
80    THREAD_4:
81      free(data_even);
82
83      RSYNC(done);
84      END_FUNCTION();
85    }
```

Figure 22: FFT Code

Line 64: We are done with our computation and send the result back to where we fetched the untransformed data. This can be overlapped with freeing the array of complex roots.

Line 67: At the bottom of the recursion we do pretty much the same. The only difference is that we know already the size of the incoming array (namely 2) and we don't need to compute special roots.

Of course, the program can be optimized in various ways. For example, we could precompute the roots or break the recursion at a certain level and then continue sequentially.

```
THREADED foo ()
{
    static int i = 1; /* not allowed */

    int x = 0;    /* not allowed */

}
```

(a) Static and Initialized Variables

```
THREADED foo (int y); /* NO */

THREADED foo (int x); /* Yes */

THREADED foo(int x)
{
  ...
}
```

(b) Prototypes

Figure 23: Examples of Illegal Use

7 Differences Between Threaded-C and ANSI C

The programmer should be aware that some standard C features are not supported in the current version of Threaded-C. Many of these features are related to the *static* and *extern* keywords. This section lists the unsupported features.

- A threaded function cannot be static.
- In the declaration of a threaded function, the `extern` specifier must be omitted.
- When defined within the scope of a threaded function, a static variable cannot be initialized. For example, in Figure 23(a), the initialization of static variable `i` is illegal.
- A threaded function is not allowed to initialize variables in their definitions. As shown in Figure 23(a), initializing variable `x` is forbidden.
- Forward declarations of threaded functions (prototypes) are supported, but in a limited fashion. If the programmer wants an ANSI prototype declaration, the parameter names must be the same as the original function declarations. For example, in Figure 23(b), using a different parameter name `y` is not allowed. However, using the same parameter name `x` is OK.

In the current version, the Threaded-C compiler does not check for the "illegal" usages above. It is up to the programmer to avoid using them.

8 Conclusion

This document has presented the current portable version of Threaded-C and the EARTH primitives used in multithreaded programs. We presented many programming examples illustrating the use of typical threaded constructs such as global addressing, use of synchronization slots, implementation of split-phase data communications, allocation of memory in remote nodes, and

implementation of other synchronization mechanisms such as mutual exclusion, all using the Threaded-C primitives.

Threaded-C is an evolving language. Our experiences writing applications in Threaded-C have allowed us to identify shortcomings in the language. Thus, a cleaner, easier-to-program version, with new features, is currently under development. As we do so, we will post these improvements on the Web. Please regularly check the following URL:

`http://www-acaps.cs.mcgill.ca/info/EARTH`

If you have suggestions, comments, or questions, please send e-mail to

`capsladm@capsl.udel.edu`

Acknowledgments

The early Threaded-C development began with the EARTH project by the ACAPS group at McGill University. The CAPSL group at the University of Delaware continues to maintain the Threaded-C system, port it to new platforms, and improve and refine it for new applications.

Threaded-C is based on the EARTH multithreaded execution model, which was proposed by Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao [6]. Herbert Hum lead the team that developed the first implementation of EARTH. The original definition of the EARTH operations was used as the basis for Threaded-C, which was designed and implemented mainly by Olivier Maquelin. He oversaw the development of both the EARTH runtime system and Threaded-C compiler.

ACAPS and CAPSL members who have contributed to the EARTH runtime system include Haiying Cai, Prasad Kakulavarapu, Cheng Li, Andres Marquez, Shashank Nemawarkar, and Kevin Theobald. Xinan Tang ported the Threaded-C compiler to the Beowulf system and currently maintains the portable version of the Threaded-C compiler. Many people have contributed to the the EARTH benchmark suite, including Nasser Elmasri, Gerd Heber, Alberto Jimenez, Olivier Maquelin, Xinan Tang, Kevin Theobald, Parimala Thulasiraman, Xinmin Tian, and Yingchun Zhu. Ruppa Thulasiram has done a lot of work to help edit this document. Finally, we would like to thank Laurie J. Hendren and her team at ACAPS for many valuable contributions and suggestions.

We would like to acknowledge the support of the Defense Advanced Research Projects Agency (DARPA), the National Security Agency (NSA), and the National Aeronautics and Space Administration (NASA) of the United States. The initial EARTH work was partially supported by the National Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 704–709, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[2] Haiying Cai. Dynamic load balancing on the EARTH-SP system. Master's thesis, McGill University, Montréal, Québec, May 1997.

[3] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 12–23, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[4] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[5] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[6] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[7] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 24(2), May 1996.

[8] Shashank S. Nemawarkar and Guang R. Gao. Measurement and modeling of EARTH-MANNA multithreaded architecture. In *Proceedings of the Fourth International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 109–114, San Jose, California, February 1–3, 1996. IEEE Computer Society TCCA and TCS.

A EARTH Primitives in Threaded-C

This appendix gives a complete list of all the EARTH Threaded-C primitives and briefly explains how they are used.

A.1 Threads and Functions

THREADED
Keyword for a threaded function declaration.

THREAD_n
Marks the beginning of a thread. The thread label within a function, n, is an integer greater than 0.

void END_THREAD(void)
Marks the end of a thread. When an END_THREAD is found, the control switches to another ready thread.

int NUM_NODES
Run-time system variable set to the number of nodes available in the system.

int NODE_ID
Run-time system variable set to the number of the local node. This number ranges from 0 to NUM_NODES - 1.

void POLL(void)
Polls the network and handles any available messages. Along with the NUM_NODES and NODE_ID primitives, this is one of the only primitives that can be used in non-threaded functions. Inserting POLL statements into long threads can significantly improve overall performance because it allows for the faster handling of external requests.

void CALL(func_name, ...)
Calls function *func_name* sequentially and blocks until that function terminates. Functions invoked with CALL must terminate with RETURN instead of END_FUNCTION.

void RETURN(void)
Ends a function that is called with the CALL primitive. This tells the compiler to generate sequential entry/exit code.

void INVOKE(int node, func_name, ...)
Start the execution of `func_name` on the node specified by the programmer and do not block. Function *func_name* must terminate with END_FUNCTION.

void TOKEN(func_name, ...)
Start the execution of `func_name` in a node selected by the runtime system. The node

is selected trying to optimize the distribution of the workload in the machine. Function *func_name* must terminate with END_FUNCTION.

void END_FUNCTION(void)
Marks the end of a threaded function that is called with INVOKE or TOKEN.

A.2 Thread Synchronization

Threads are often associated with a synchronization slot. The sync count in that slot represents how many signals the thread has to wait for before it can be activated. The programmer can initialize the sync count and update the count to control the firing of a thread. We use the following EARTH primitives to operate on the sync slots:

SLOT
A pre-defined type used to allocate the synchronization slots that will be used in the function.

SLOT SYNC_SLOTS[N]
This is the declaration of synchronization slots using the pre-defined type SLOT. This declaration must appear at the beginning of a function.

SPTR
A pre-defined type for synchronization pointers. It is defined as: **typedef SLOT *GLOBAL SPTR**.

void *GLOBAL FRAME_ADR(void)
Returns a global pointer to the current frame.

void *IP_ADR(int thread_num)
Returns a (local) pointer to the first instruction of thread *thread_num*. Notice that the pointer returned does not need to be a global pointer because each node has a copy of the program code loaded at the same address.

SPTR SLOT_ADR(int slot_num)
Returns a global pointer to sync slot *slot_num*.

void INIT_SYNC(int slot_num, int init_count, int reset_count, int thread_num)
Initializes sync slot *slot_num* with the initial counter value *init_count*, the reset value *reset_count*, and a thread pointer for thread *thread_num*.

void SYNC(int slot_num)
Decreases the sync count of slot *slot_num* by one. If the count reaches zero the corresponding thread is scheduled for execution.

void RSYNC(SPTR slot_adr)
Same as SYNC(), but the sync slot is specified by a global address.

void INCR_SYNC(int slot_num, int val)
Increases the sync count of slot *slot_num* by *val*. If the count becomes zero the corresponding thread is scheduled for execution.

void INCR_RSYNC(SPTR slot_adr, int val)
Same as INCR_SYNC(), but the sync slot is specified by a global address.

void SPAWN(int thread_num)
Schedules local thread *thread_num* for execution.

void RSPAWN(void *GLOBAL FP, void *IP)
Schedules local thread *thread_num* for execution. The thread is specified through its frame and instruction pointers.

Implicit sync operation
All data transfer primitives also perform a sync operation after the data has reached its destination.

A.3 Data Transfer Primitives

The data transfer primitives support remote memory accesses and block data transfers. Short data transfers of single bytes or words of memory are supported by the GET_SYNC_x and DATA_SYNC_x. Several versions of these primitives exist, which are distinguished by their suffix. For example, the suffix **_L** is used for 32-bit (long word) values. Here is the complete list of suffixes:

_B (char): Single byte (8 bits).

_S (short): Short word (16 bits).

_L (long): Long word (32 bits).

_F (float): Float size (32 bits).

_D (double): Double size (64 bits).

_G (void *GLOBAL): Global pointer (either 32 or 64 bits).

In addition, the sync slot that should be signaled when the operation terminates can either be specified as a (local) slot number (_SYNC_ variants), or as a global pointer (_RSYNC_ variants). Here are the basic communication primitives:

void DATA_SYNC_x(T datum, void *dest, int slot_num)
Sends a value to the destination address and then update the specified sync slot. The type of the value has to be either a byte, a short, a long, a float, a double, or a global pointer.

void DATA_RSYNC_x(T datum, void *dest, SPTR slot_adr)
Same as DATA_SYNC_x(), but the sync slot is specified as a global address.

void GET_SYNC_x(void *GLOBAL src, void *GLOBAL dest, int slot_num)
Reads a value from the source address and copy it to the destination address. Then, update the specified sync slot.

void GET_RSYNC_x(void *GLOBAL src, void *GLOBAL dest, SPTR slot_adr)
Reads a value from the source address and copy it to the destination address. Then, update the specified sync slot. The sync slot is specified as a global address.

void BLKMOV_SYNC(void *GLOBAL src, void *GLOBAL dest, long length, int slot_num)
Copies *length* bytes of data from the source to the destination address and updates the specified sync slot.

void BLKMOV_RSYNC(void *GLOBAL src, void *GLOBAL dest, long length, SPTR slot_adr)
Same as BLKMOV_SYNC(), but the sync slot is specified as a global address.

A.4 Global Address Support

GLOBAL
Type qualifier used to distinguish global pointers (64-bit entities) from local (normal) pointers.

T *GLOBAL TO_GLOBAL(T *ptr)
Turns a local pointer into a global pointer that points to address `ptr` on the local node. In the portable implementation the type of the result depends on the type of the argument. On MANNA, the result is of type pointer to void.

T *TO_LOCAL(T *GLOBAL gptr)
Turns a global pointer into a local pointer (extracts the address part of a global pointer). Note that it is possible to dereference a global pointer without first turning it into a local pointer. On MANNA, the result is of type pointer to void.

T *GLOBAL MAKE_GPTR(T *ptr, int node)
Takes a node number and a local address and returns the corresponding global pointer. On MANNA, the result is of type pointer to void.

int OWNER_OF(T *GLOBAL gptr)
Returns the node pointed to by *gptr* (extracts the node part of a global pointer).

int IS_OWNER(T *GLOBAL gptr)
Returns *true* if *gptr* points to the local node.