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

Threaded-C Language Reference Manual (Release 2.0)

Guy Tremblay †
Kevin B. Theobald
Christopher J. Morrone
Mark D. Butala
José Nelson Amaral ‡
Guang R. Gao

CAPSL Technical Memo 39
September 2000

†Dépt. d'informatique, Université du Québec, Montréal, Québec, Canada
‡Computer Science Dept., University of Alberta, Edmonton, Alberta, Canada

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

This document describes a revised version (release 2.0) of 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 sequence of small examples to acquaint the reader with this new programming language and style. This tutorial is followed by two detailed examples, one showing how locks typically used for mutual exclusion in shared-memory style programming can be implemented with the constructs of Threaded-C, and the other a program computing a one-dimensional FFT. The appendixes contain a list of all Threaded-C commands and their usage together with a brief description of some useful library modules.

Contents

List of Figures

1 Introduction

In multithreaded systems based on the EARTH execution model [5, 6, 7], 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 a revised version of Threaded-C, which builds on a previous description presented in [12].

Several dialects of Threaded-C have been developed. The first version, developed specifically for the MANNA implementation of EARTH, was tuned to the MANNA architecture, and its compiler could only generate code for the MANNA [11]. 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 [3]. Therefore, a second version of Threaded-C was developed to be portable across all platforms, although this version has a slightly more complex syntax.

This first portable version of Threaded-C has been available on several platforms. The most generic implementation could 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 when this first version of Threaded-C was described [12], the platforms supported included the IBM SP-2 and SP-3, the MANNA, and a network of Sun workstations connected with Myrinet. It was also possible to run Threaded-C code sequentially on a Sun or RS6000 workstation.

In the first portable version of Threaded-C mentioned above, slots and fibers could only be identified using explicit (hard-wired) numbers. Thus, although based on C, old style Threaded-C programs sometimes were as difficult to understand as some assembly language programs — in fact, in some aspects, Threaded-C programs could be seen as *lower-level* than assembly language programs, because the latter make it possible to use symbolic labels. Different instructions also had to be used to terminate execution of a threaded function depending on whether this function had been `INVOKE`d or `CALL`ed. In other words, the way to terminate a function was dependent on its calling context, which is clearly a strong form of coupling and leads to less reusable procedural abstractions. A revised version of Threaded-C fixing these and other problems was subsequently developed [1]; this version, however, was never made widely public.

Another important restriction of these previous versions of Threaded-C is that they assumed that a single fiber would be executing on a processor at any given time. For example, CAPSL Memo 19 [12] presented a mutual exclusion example which, in order to work correctly, required that a single fiber be executing on the processor owning the shared value. Because SMP implementations of EARTH and Threaded-C are becoming available, this assumption clearly is not valid anymore and appropriate extensions to the language became necessary in order to correctly support mutual exclusion.

This document describes the newly revised version of the portable Threaded-C. The next two sections review the salient features of EARTH and Threaded-C (these have been covered in more detail in prior documents). The next section covers the history and evolution of Threaded-C. 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 a description of the Threaded-C language. Section 4 is an introduction to the portable Threaded-C language (release 2.0), using examples to illustrate its primary features. The section that follows discusses some advanced features of the language. Section 6 shows two slightly larger examples of Threaded-C code, one demonstrating how to implement locks that can be used to ensure mutual exclusion between fibers and the other computing a one-dimensional FFT. Programmers should take note of Section 7, which briefly mentions a few standard C features not currently 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 multi-threading.[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 fibers 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 [2], a multiprocessor consisting of dual-processor nodes. In each MANNA node, one processor performs the EU's task of executing fibers, 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 [4], or ordinary sequential C.

2.1 Variations of Threaded-C

The EARTH operations [5, 6, 7] 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), so that regular pointers might not be big enough to accommodate global addresses. Furthermore, most parallel machines do not 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.[2]

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, do not have a second processor, and on these machines, it is necessary to perform the SU functions on the same processor that runs the application fibers. How are incoming messages handled in this case? Experiments with a single-processor-node implementation of EARTH [10] 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 fiber 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.)

At this point in its evolution, Threaded-C was still very much tied to its machine-language-like origin. For example, slots (and fibers) were strictly identified using numbers that had a direct correspondence with the underlying EARTH synchronization slots, instead of being names using symbolic identifiers as done in modern high-level languages. Furthermore, this mapping had to be managed explicitly by the programmer. Thus, a typical threaded function with two fibers might have looked as shown in Figure 1, where the use of hard-wired number sometimes made programs slightly difficult to understand.[3]

Identifying slots and fibers with symbolic names became possible with Threaded-C release 1.1 [1], which also fixed a number of annoying features and restrictions that were present in the initial portable Threaded-C release. However, although it brought a number of interesting changes, this revised version

[2]Note that such a global pointer, also known as a *global handle*, does not have anything to do with an ordinary C global variable. Recall that, in C, a global variable is one which is defined *outside* the scope of a function definition. On the other hand, a global handle is a pointer that refers to some space in the global memory of the machine, space which can be associated with some local (stack) variable, global variable or heap variable. The notion of global handle in the context of Threaded-C should thus be interpreted as meaning *possibly remote*.

[3]Formerly, fibers used to be called threads, thus the use of `THREAD_`n and `END_THREAD()` in this example. Note that although it was possible to use `define` to introduce symbolic names for slots, it was difficult to do the same for thread names, because of the "`THREAD_k`" syntax.

```
THREADED foo( ... )
{
  SLOT SYNC_SLOTS[2];

  INIT_SYNC( 0, 1, 1, 1 );
  INIT_SYNC( 1, 1, 1, 2 );

  SYNC(1);
  INVOKE( ..., SLOT_ADR(0), SLOT_ADR(1) );

  THREAD_1:
    ...
    END_THREAD();

  THREAD_2:
    ...
    END_FUNCTION();
}
```

Figure 1: A Small Threaded Function with 2 Fibers

of Threaded-C never really became widely public.

The new version of the portable Threaded-C (release 2.0) described in the present document initially grew out of the CAPSL group's desire to make public an open source version of both the Threaded-C compiler and the new SMP implementation of the run-time system (RTS). Having an SMP implementation means that a critical assumption made in earlier descriptions of the language was not valid anymore, where it was always assumed that at most a single fiber could be executing on a node at any given time. Going to an SMP implementation implies that it now becomes possible to have multiple concurrently executing fibers on the same node. In turn, this implies that some form of mechanism now has to be provided to enforce mutual exclusion between fibers executing, on the same node, some critical sections. As will be described in section 5.4, in the context of the data flow style of programming encouraged by Threaded-C, only two mechanisms needed to be added to the language for this purpose: mutually exclusive fibers and atomic mailboxes.

The changes made to Threaded-C in order to provide a sound language for SMP implementations have not been the only modifications made to the development of Threaded-C release 2.0. A number of modifications have also been made in order that Threaded-C be more faithful to its standard C backbone, e.g., allowing pointer arithmetic on global handles, allowing local variable declarations in inner blocks, etc. Threaded-C release 2.0 also introduces a number of modifications and extensions whose goal is to make Threaded-C a cleaner and easier to use programming language, e.g., more uniform convention for naming instructions, implicit binding of synchronization slots to fibers with initialization/reset counts specified locally, overloaded data and synchronization operations that lead to a drastic reduction in the total number of different instructions (4 instead of 28), a new syntax for defining fibers. All these features of this new release of Threaded-C will be introduced through examples presented in subsequent sections. For a more detailed presentation of the rationale that lead to the current language design, including some alternatives which were examined but rejected, the reader should consult CAPSL Technical Note 09 [13].

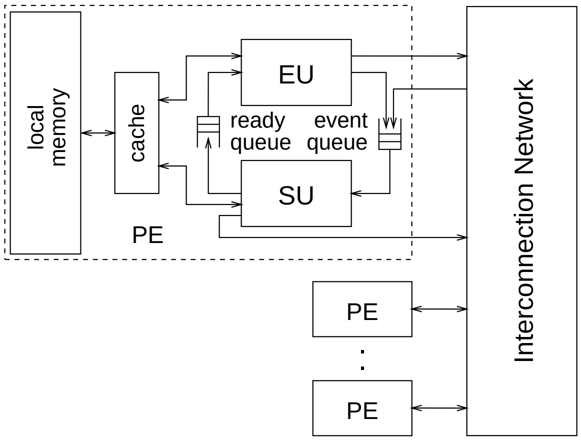

Figure 2: EARTH Architecture

3 The EARTH Programming Model

In the EARTH programming model, fibers (formerly called threads) are sequences of instructions belonging to an enclosing function. Fibers always run to completion — they are non-preemptive. Synchronization mechanisms are used to determine when fibers become executable (or ready). Although it is possible to spawn a fiber explicitly, in most cases a fiber 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 *fiber id*, of the fiber that is to be started when the counter reaches zero. This mechanism permits the implementation of dataflow-like firing rules for fibers (a fiber 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.[4] Each EARTH node has an *Execution Unit* (EU) and a *Synchronization Unit* (SU) linked to each other by queues (see Figure 2). The EU executes active fibers, and the SU handles the synchronization and scheduling of fibers and communication with remote processors.

This division allows the implementation of multithreading architectures with off-the-shelf microprocessors mass-produced for uniprocessor workstations [7]. The EU is expected to be a conventional microprocessor executing fibers sequentially.[5] The SU performs specialized tasks and is relatively simple compared to the EU. Thus, the SU can be implemented in a small 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 2 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 fibers which are ready to be executed. The EU fetches a fiber from the ready queue whenever the EU is ready to begin executing a

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

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

new fiber. 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 fiber, after the load has completed.

3.2 Fibers

A fiber is a non-preemptive, atomically-scheduled sequence of instructions. When an EU executes a fiber, it executes the instructions according to their sequential semantics. In other words, instructions within a fiber 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 fiber. Both conditional and unconditional branches are only allowed to destinations within the same fiber.

EARTH fibers are *non-preemptive.* Once a fiber begins execution, it remains active in the EU until it executes an EARTH operation to terminate the fiber. If the CPU should stall (e.g., due to a cache miss), the fiber will not be swapped out of the EU. There are no mechanisms to check that data accessed by an executing fiber is actually valid or to suspend the fiber 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 fiber begins execution. 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 *sync count,* a *reset count,* and a *fiber pointer*, as shown in Figure 3. The sync count indicates the number of sync signals that have to be received by the sync slot before the corresponding fiber can be enabled. When a sync signal is received, the sync count is decremented. If the count reaches 0 the fiber specified by the fiber pointer is placed in the ready queue and the sync count is set back to the value of the reset count. The fiber pointer (or *fiber id*) consists of a frame pointer

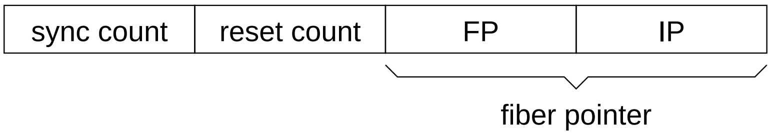

Figure 3: Synchronization Slot Format

(*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 fiber.

For instance, consider the issue of a split-phase *read* operation in fiber i. After issuing the read operation, fiber 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 fiber, fiber j. To ensure the correctness of the split-phase read, fiber j is associated with a sync slot; of course, fiber i also has to specify that fiber j, more precisely its sync slot, is the one that should be signaled when the data is available. This way, fiber j will not be scheduled for execution until a sync signal is received signaling that the remote read has finished.

3.2.2 Fiber Scheduling

The only restrictions imposed by the EARTH model on fiber scheduling are the synchronization rules implemented by the sync slots. A fiber 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 fibers are placed at the end of the ready queue and the EU always read fibers from the beginning of the ready queue. However, more elaborate policies are possible in implementations that allow random access to the ready queue [7]. For instance, fibers can be prioritized to favor fibers known to be on critical paths, or fibers can be scheduled in LIFO order to benefit from register locality.

The atomicity and non-preemptiveness of fibers does not prevent multiple fibers from running simultaneously on the same EARTH node. The EARTH model allows for the EU to maintain multiple independent active fibers, 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 fiber execution.[6]

3.3 Threaded Functions

A fiber is always part of an enclosing *threaded function.* A threaded function is a C function containing local state (function parameters, local variables, and sync slots) and one or more fibers of code. All fibers 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 fibers access this frame through its associated *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, function 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

[6] However, there does exist a language feature — the `EXCLUSIVE` annotation described in section 5.4 — that makes it possible to indicate that a fiber should never be scheduled concurrently with another fiber of the same function.

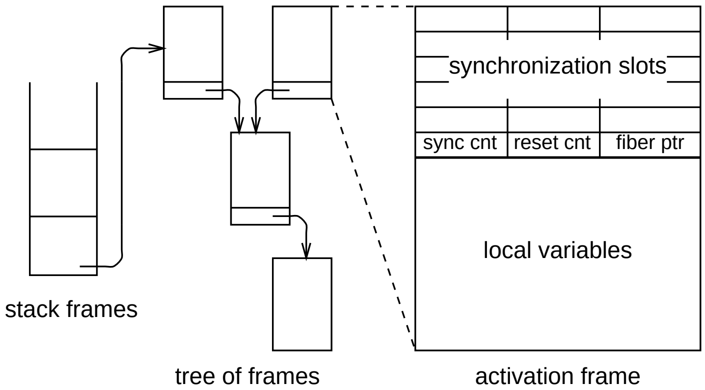

Figure 4: Stack and Activation Frames

from a heap. Figure 4 illustrates parent-children relationships between function invocations, typical of divide-and-conquer parallelism, thus leading to a tree of frames — in general, arbitrary relationships are possible, since Threaded-C allows any kind of relationship to be established between threaded functions. 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 fiber execution.

Since all fibers 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 [7] 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 fibers 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 4. 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 fibers, can be invoked in parallel, and can use all the Threaded-C language extensions. A *sequential function* is a standard function without any multithreading extensions. 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 4 shows a stack that stores the sequential function frames.

[7] Note that, to be more precise, in the current SMP implementation, the caller can choose the processing node on which a threaded function is to be executed, but not the specific processor of the selected node on which the function is to be executed.

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

Figure 5: A Simple Threaded-C Function

4 Introduction to Threaded-C

This section introduces the basic features of Threaded-C and illustrates their use through 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. Execution of the `MAIN` function, or any other threaded function, is terminated with the `TERMINATE` primitive.

One of the simplest `MAIN` functions is shown in Figure 5. 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 except for the fact that they cannot introduce fibers and, thus, cannot use operations related with the manipulation of local sync slots or operations that involve terminating the current fiber or threaded function.

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.

In the example above, regardless of how many nodes run the program, there will be only one "`Hello World!`" string printed. In contrast to many parallel programming languages, 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 as 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, which is why its return type is simply indicated as THREADED. Threaded functions, as mentioned earlier, must end with the TERMINATE primitive.

Input/output in Threaded-C is done using the standard C operations, such as scanf, printf, fprintf, etc. When a program is running on a number of different nodes, each node will generally have its own private stdout stream, which is where printf output is sent by default.[8] If we print a string on another node, for example using printf, then we might like to include some information about the node itself, to indicate explicitly which node produced that output. The Threaded-C language defines two integer constants related with this:

- NUM_NODES — the number of nodes that are executing the program;
- NODE_ID — the identity (natural 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 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.[9]

With the explanations given above, our first attempt to printing multiple copies of the "Hello World!" might look like Figure 6. The output that we observe for this program will depend on the number of nodes on which we are running the program. Even for a specific number of nodes, however, the output might vary between runs. For instance, suppose that the output of all the nodes are merged and sent to a single terminal. In this case the order in which the nodes will print their output may change from one run to the next.

More importantly, what might also be noticed is that some of the outputs may not be printed at all. The reason is that the fiber 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 then continue its execution. The next action of the MAIN function, in line 16, is to terminate the execution of the threaded function activation. As a consequence the MAIN function might terminate *before* all the invoked print_hello functions are able to print. To fix this behavior, we need a *synchronization mechanism* which allows us to impose some order over threads and fibers 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.

[8] In the current SMP implementation, the default is for the output of all processors on a given node to be merged onto a common stdout stream. A compiler option, however, allows the program output produced by each processor to be sent to an output stream unique to each processor.

[9] Note that, in the current SMP implementation, it is not possible to determine on which processor a threaded function is being executed. Only the node number, shared by all processors on that node, is available.

```
#include <stdio.h>

THREADED print_hello( void )
{
   printf( "From node %d: Hello World!\n", NODE_ID );
   TERMINATE;
}

THREADED MAIN( int argc, char* argv[] )
{
  int i;

  for ( i = 0; i < NUM_NODES; i++ )
    INVOKE( i, print_hello );

  TERMINATE;
}
```

Figure 6: Hello World on All Nodes

A *fiber* in the context of EARTH multithreading is a sequence of instructions appearing within the scope of a threaded function and whose execution order is determined by the standard C flow of control. Fibers run to completion, i.e., they are *non-preemptive.* Fibers without dependences may run concurrently, and fibers may generate and activate other fibers and threaded functions through various mechanisms to be presented.

A function such as `print_hello` in Figure 6 consists of a single fiber, which gets scheduled for execution as soon as a frame for this function has been allocated by the runtime system.[10] If more fibers are desired, they must be indicated with the `FIBER` keyword. The semantics for such fibers is that the standard C flow of control is followed until a `FIBER` keyword is encountered, at which point the currently executing fiber stops executing and another fiber is selected for execution. In other words, there is an *implicit* `END_FIBER` instruction — instruction used to switch to another fiber — immediately before each `FIBER` keyword.[11]

If a function has n fibers, all fibers but the first one begin with the `FIBER` keyword; this first fiber simply begins with the first executable statement of the function, and does requires any explicit `FIBER` keyword. The first (unlabeled) fiber is sometimes called the *initialization fiber* because it automatically is eligible to begin execution once the function frame has been set up. No other fibers in this function instance can start running until enabled by some appropriate synchronization events.

Synchronization in this context is the mechanism that allows us to impose some global order on fiber execution. For instance, if we have two fibers, `FIBER_1` and `FIBER_2`, with a data dependence that demands that `FIBER_1` executes before `FIBER_2`, then we must use synchronization to guarantee that `FIBER_2`'s execution does not start before the execution of `FIBER_1` ends.

In Threaded-C, synchronization is handled through *synchronization slots.* A sync slot has three fields: a sync count, a reset count, and an associated fiber identifier. The sync count changes dynamically, while the other two fields remain constant. The sync count holds the number of synchronizations that must

[10] Recall that a fiber may be ready and scheduled for execution, but may still be waiting for the execution unit to become available.

[11] Note that there is also an implicit `END_FIBER` just before the function's closing bracket: if the normal flow of control reaches the textual end of the function, it is only the current fiber's execution which is stopped; a function activation is never terminated implicitly and must always be terminated explicitly using the `TERMINATE` instruction.

```
1  #include <stdio.h>
2
3  THREADED print_hello( SPTR done )
4  {
5     printf( "From node %d: Hello World!\n", NODE_ID );
6     SYNC(done);
7     TERMINATE;
8  }
9
10 THREADED MAIN( int argc, char* argv[] )
11 {
12    int i;
13
14    for ( i = 0; i < NUM_NODES; i++ )
15      INVOKE( i, print_hello, TO_SPTR(ALL_DONE) );
16
17    FIBER ALL_DONE <* NUM_NODES *> {
18      TERMINATE;
19    }
20 }
```

Figure 7: Safe Hello World

be performed before the fiber can be executed. Each incoming synchronization signal decrements the sync count by one. Once the sync count reaches zero, the corresponding fiber is activated, and the sync value is updated with the reset value.

The reset count, the *initial* value of the sync count and the binding between a sync slot and its corresponding fiber can generally be set in two different ways: explicitly, using the `INIT_SLOT` command, or implicitly, using the `<*...*>` syntax for fibers described below. Note that in all but one example in this tutorial, the reset count is irrelevant, as the sync slot is only used once. However, real applications are likely to synchronize a fiber repeatedly, in which case it is important that the reset count has a proper value. The initial value of the sync count ("init count") is usually the same as the reset count. However, it may be different, for example, if the first activation of a fiber has special synchronization requirements.

As alluded above, starting with Threaded-C release 2.0, it is possible to have implicit definitions of the sync slots, without the need for any explicit `INIT_SLOT` instructions. In other words, contrary to the previous versions of Threaded-C, in many cases the programmer can let the compiler take care of declaring the sync slots and binding them to their associated fibers. As we will see in the next example, this is done by automatically associating a sync slot with each fiber (except for the initialization fiber, which does not need any slot) and by specifying the appropriate init and reset counts right next to the fiber definition.

A *synchronized* version of our hello-world program with such implicit slots would look as described in Figure 7:

Line 3: The `print_hello` function takes a reference to a sync slot as argument: `SPTR` (Slot PoinTeR) is a pre-defined Threaded-C data type for a pointer (global handle) to a sync slot.

Line 6: After `print_hello` finishes printing, it sends a synchronization signal to the sync slot it received as argument (`done`).

```
#include <stdio.h>

THREADED print_hello( SPTR done )
{
    printf( "From node %d: Hello World!\n", NODE_ID );
    SYNC(done);
    TERMINATE;
}

THREADED MAIN( int argc, char* argv[] )
{
   int i;

   INIT_SYNC( ALL_DONE, NUM_NODES, NUM_NODES, ALL_DONE );

   for ( i = 0; i < NUM_NODES; i++ )
     INVOKE( i, print_hello, TO_SPTR(ALL_DONE) );
   END_FIBER;

   FIBER ALL_DONE
     TERMINATE;
}
```

Figure 8: Safe Hello World with Explicit Sync Slots Specification

Line 14: This loop is a simple example of a *forall* loop: parallel activations of `print_hello` are created on each of the `NUM_NODES` node.

Line 15: `TO_SPTR(`*slotName*`)` is a built-in command generating a pointer (of type `SPTR`) to the sync slot *slotName*. This is what is assigned to the argument `done` (Line 3). The next item describes where this sync slot comes from.

Line 17: After executing the `for` loop, following the normal C flow of control brings the program execution to encountering the `FIBER` keyword; this automatically terminates the currently executing fiber (the initialization fiber) — there is an implicit `END_FIBER`, which terminates the currently executing fiber, before all `FIBER` keywords.

Fiber `ALL_DONE`, like all explicit fibers, is not executed through the normal flow of control; rather, execution of `ALL_DONE` will be triggered when some sync slot, bound to this fiber, will have received enough signals. In the present case, the fiber name (which follows the `FIBER` keyword) is followed by an init count specification (`<* NUM_NODES *>`) and there is no explicit `INIT_SLOT` command for a slot with the same name. Thus, there is an implicitly declared sync slot also called `ALL_DONE` which can be used to send signals to this fiber. This explains why, in Line 15, it is correct to send the argument `SPTR(ALL_DONE)` to the invoked functions: the `ALL_DONE` identifier in this expression simply refers to the implicitly defined sync slot for fiber `ALL_DONE`.

Line 18: Since fiber `ALL_DONE` is waiting for a signal from each of the function invocations created in Line 15 (the count specified in Line 17 is `NUM_NODES`), it thus plays the role of a *barrier*. It will not run until all nodes have sent their termination signal through slot `done` and, thus, will not terminate until every node has finished printing its output.

Note that when a fiber and an implicitly associated sync slot is defined in the way illustrated in our preceding example, it is possible to specify both the initial and reset counts associated with the sync slot. When only one value is specified, as in our example, then the reset count is assumed to be the same as the initial count; the reset count , if it differs, can simply be written after the init count as in the following:

```
FIBER fib_name <* init, reset *>
```

It is also important to note that no special semantics is associated with the opening and closing brackets appearing, in our example, after the fiber heading for `ALL_DONE`. As mentioned earlier, the termination of the current fiber is associated with the normal flow of control reaching a `FIBER` keyword. Thus, by making explicit both the fiber termination and sync slots declaration and by eliminating the unnecessary brackets from the example of Figure 7, the equivalent program presented in Figure 8 would then be obtained — this latter version should be familiar for programmers who learned Threaded-C from CAPSL Technical Memo 19 [12].

4.3.1 Local and Remote Synchronization

Threaded-C provides two flavors of synchronization: local and remote. Both are indicated using the same `SYNC` command. The preceding example shows a remote synchronization, which takes a global handle to a sync slot (`SPTR`) as its argument. There is also a local form of `SYNC`, which can be used for synchronization *within* a function invocation. `SYNC(`*slotId*`)` (where *slotId* is a local slot identifier) sends a synchronization signal to the indicated local sync slot.

It is important to note that, in this context, "remote" does not necessarily 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. As mentioned earlier, such an `SPTR` is obtained from a simple local slot identifier using `TO_SPTR`. Of course, a `SYNC` performed on an `SPTR` can also refer to a local slot; thus, in the following piece of code, two signals would be sent to `FIB1`:

```
...
SYNC( FIB1 );
SYNC( TO_SPTR(FIB1) );

FIBER FIB1 <* init, reset *> ...
```

4.3.2 Spawning Fibers

Threaded-C provides an alternate mechanism for starting fibers. A fiber may be started directly using a `SPAWN` command. Such a command specifies a particular fiber, 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`); rather, it is equivalent to a fork operation as found in many parallel programming languages. If a spawn occurs in the middle of a fiber, the fiber performing the spawn continues to execute and the new fiber could begin execution concurrently.

As with sync operations, the spawn command can receive either a local fiber id or a remote fiber pointer. The local `SPAWN` command simply takes a fiber name as argument, which refers to a fiber in the local function. The remote variant is described in appendix A.

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 full knowledge 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 (Figure 7), we could replace line 15 with the following line

```
TOKEN( print_hello, TO_SPTR(ALL_DONE) );
```

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.[12]

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, which include both put-sync and get-sync) operators. The EARTH system guarantees the *atomicity* of these operators in the sense that in a combined data-sync operation, the data transfer is guaranteed to be complete before the synchronization is performed.

A data transfer operation needs some location where to put the data to be transmitted. 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.[13] Therefore, Threaded-C uses *global addresses*, also called *global handles*, for EARTH operations between nodes.

A global handle is like a regular pointer, but is distinguished by the keyword `GLOBAL`. For instance, "`int *GLOBAL g`" declares `g` to be a global handle to a location, possibly on a remote node, containing an `int`. A global handle is attached to a specific node, so it is possible to determine the node ID owning a specific handle. Since this is implementation-dependent, special functions are provided for this, e.g., `OWNER_OF(g)` would return the node owning the location refered by `g`, `IS_LOCAL(g)` would determine if `g` refers to a location in the local memory (other functions are listed in appendix A). Programmers

[12] 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.

[13] Anyhow, on some machines, the combined memory of all the processors may be too large to be addressed by regular pointers.

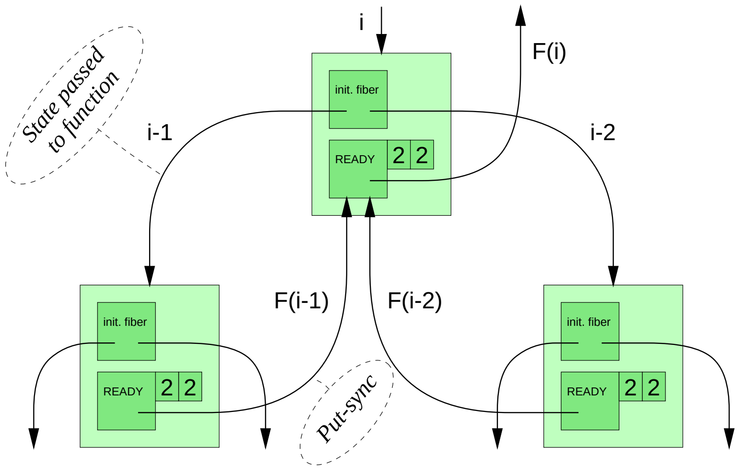

Figure 9: Fibonacci Code Structure

should use these operations and not make any assumptions about the internal representations of these pointers.[14]

It is important to stress that one cannot dereference a global handle directly. An operation such as `*g`, where `g` is declared as above, is *illegal*. This is why this language construct has been called global *handle* instead of global *pointer*, to prevent programmers from making unwarranted assumptions about what they can do with such handles. On the other hand, standard pointer arithmetic is allowed on global handle. Also, one can convert explicitly between local pointers and global handles using EARTH operators, as shown in upcoming examples and in appendix A.

4.5.1 Simple Data Synchronization

The following example illustrates global handles and data-sync operations, more precisely, put-sync operations. We wish to compute Fibonacci numbers using a naïve recursive approach. If the recursive function `fib`(i) determines that it is not a leaf, it calls `fib`$(i-1)$ and `fib`$(i-2)$, adding their results. Because load 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 fibers, like the `MAIN` function in our hello-world example, since the summing of the results is dependent on data sent back by the tokens, viz., both results must have been received before their sum can be computed.

The linkage of one function instance with its children is shown in Figure 9. The diagram shows each function instance as a light box, with two darker boxes representing the two fibers. Box "`init. fiber`" in each function represents the initialization fiber; the other box is fiber `READY` (see Figure 10, Line 16). Each fiber (except initialization fibers) shows the init and reset counts for the sync slot that controls that fiber.[15]

The code for the Fibonacci recursive implementation is shown in Figure 10. The following explains the new commands introduced in this example:

[14]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 a no-op. However, programmers should always use the `GLOBAL` keyword to ensure their code is portable.

[15]It is assumed that each fiber is controlled by a single sync slot. This is usually true in most programs, but it *is* legal to have more than one sync slot control the same fiber. In this case, explicit `INIT_SLOT` instructions must be used.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  THREADED fib( int n, int *GLOBAL result, SPTR done )
5  {
6    int r1, r2;
7
8    if (n <= 1) {
9      r1 = 0; r2 = 1;
10     SPAWN( READY );
11   } else {
12     TOKEN( fib, n-1, TO_GLOBAL(&r1), TO_SPTR(READY) );
13     TOKEN( fib, n-2, TO_GLOBAL(&r2), TO_SPTR(READY) );
14   }
15
16   FIBER READY <* 2 *> {
17     PUT_SYNC( r1 + r2, result, done );
18     TERMINATE;
19   }
20 }
21
22 THREADED MAIN( int argc, char* argv[] )
23 {
24   int n, res;
25
26   n = atoi(argv[1]);
27   TOKEN( fib, n, TO_GLOBAL(&res), TO_SPTR(FIB_DONE) );
28
29   FIBER FIB_DONE <* 1 *> {
30     printf( "fib(%d) = %d\n", n, res );
31     TERMINATE;
32   }
33 }
```

Figure 10: Recursive Fibonacci

Line 4: The parameters to the `fib` threaded function include a global handle (`result`) which will receive the result of the function (the n^{th} Fibonacci number); they also include a slot to be signaled when that result has been computed.

Line 6: Local variables `r1` and `r2` are introduced to receive the results of the recursive calls.

Lines 8–10: This function call is a leaf (no more recursion). Write the values 0 and 1 into `r1` and `r2` and directly spawn fiber `READY`.

Lines 11–13: 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 addresses `&r1` and `&r2` into global handles so that the two instances of `fib` have places to send their results. Note that both calls receive the same sync slot (`TO_SPTR(READY)`), and that this fiber require 2 sync signals (Line 16) before it can start.

Lines 16–19: Fiber `READY` runs after two signals have been received in sync slot `READY`, indicating that both results are ready in `r1` and `r2` because both children (Lines 12–13) have sent back their result. Fiber `READY` may also be spawned directly (Line 10) if this invocation of the function `fib` is a leaf of the recursion tree. Add the sub-results together and use a `PUT_SYNC` (on a remote slot) to send the sum to the caller.

Line 22: The `MAIN` function is similar to the `MAIN` function in the hello-world example, except that it does some command-line processing (Line 26) to determine the value to be used as argument to the call for `fib`. Note that only one sync signal needs to be received (Line 29) before the computed result can be printed since `MAIN` invokes `fib` only once (Line 27, using the `TOKEN` mechanism).

4.5.2 Fetching Data with Synchronization

The Fibonacci example used put-sync commands 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 cases where the situation is reversed: the producer does not 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 fiber during this time to hide the operation's 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 fibers.

Symetric to the `PUT_SYNC` operation, there is a corresponding `GET_SYNC` operation. For instance, the `PUT_SYNC` instruction in line 17 (Figure 10) sends the first argument (an `int`) to the global handle `result` (which points to an `int`) and then sends a signal to the synchronization slot indicated by the global handle `done`. An example of a symmetrical `GET_SYNC` operation would be:

```
GET_SYNC( remote, TO_GLOBAL(&local), SOME_SLOT );
```

(assuming `remote` is a global handle pointing to an `int`, and `local` is a local `int`). This instruction would send a request to the node owning global handle `remote`, requesting that the contents of that location be sent to `local`, and that the local sync slot `SOME_SLOT` 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 fiber guarded by slot `SOME_SLOT` will use the value in `local`.

The next section shows a program example that uses a get-sync operation. As with simple syncs (`SYNC` operation), the `PUT/GET_SYNC` operations can be used either with a local slot (local slot identifier) or with a global handle pointing to a slot (`SPTR`).

Note that these data-sync operations can be used with arguments of any type, including `struct`; what then gets transmitted is the value that would have been passed to a regular or threaded function call (call-by-value semantics). Of course, this means that the following `PUT_SYNC` only transmits the starting address of array `a`, not the whole array:

```
int a[100];
...
PUT_SYNC( a, dest, done );
```

The following section describes how to move large block of data.

4.5.3 Block Moves

The data-sync (put-sync and get-sync) primitives only move a single datum. Threaded-C provides an explicit `BLKMOV_SYNC` operation to move contiguous blocks of arbitrary size. As with the `GET_SYNC` operation, a `BLKMOV_SYNC` operation takes two data addresses (global handles) and a sync slot (either local or remote). An additional argument specifies the size of the block of data to be moved. Block

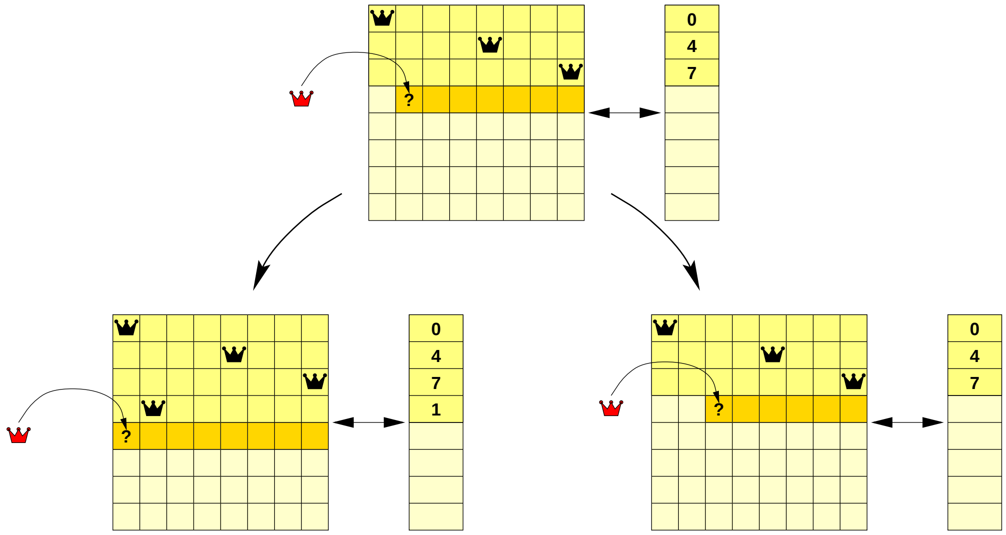

Figure 11: Recursion in N-Queens

moves are thus similar to the `memcpy()` function in C: addresses are generic (untyped), and the size argument specifies the size in bytes.[16]

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.*[17]

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 function 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 function 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 to place 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 11:

[16] 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 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.

[17] Note that there is another variant of `BLKMOV_SYNC`, described in appendix A, which must receive two sync slots: one to signal when the source has been read and can be overwritten, and one to signal when the destination has been written.

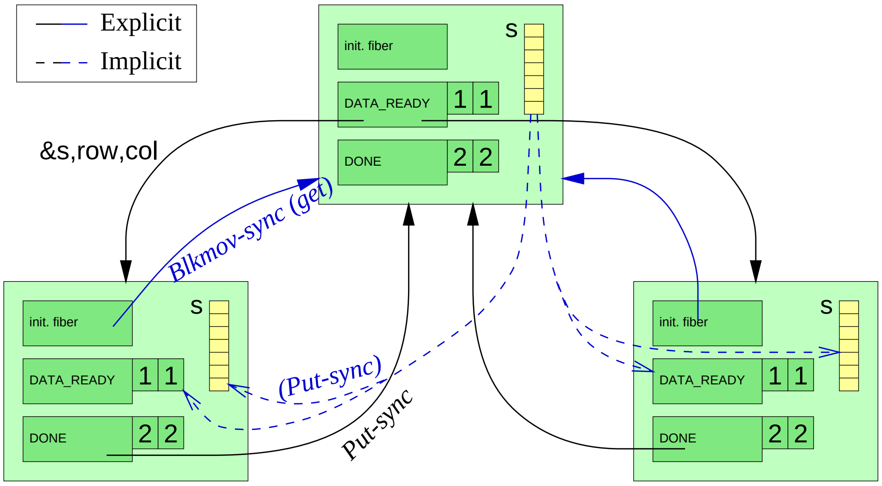

Figure 12: Data Transfers and Synchronization in the N-Queens Solution

- 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 program calls the search function with an empty board ($r = c = 0$).

The control structure is similar to our Fibonacci example, suggesting that a code structure similar to Figure 9 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 cannot 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 cannot be passed "by value" in C, the partial solution cannot be directly passed to the next level in the parameter list.

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 fiber in Figure 9, which performs the recursion, into two fibers: 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 (Figure 13), the initialization fiber issues the block move request, and then fiber `DATA_RECEIVED` is signaled when the data becomes available. This fiber then invokes the new search functions (using `TOKEN` commands), and these functions must synchronize fiber `DONE` to signal their completion. The structure is illustrated in Figure 12, 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 13 lists the recursive search routine. The program assumes a maximum of 24 queens (`MAX_BOARD_SIZE`: see Figure 14). 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()` together with an auxiliary error routine, is listed in Figure 14.

Line 16 of Figure 13 asks for a copy of the remote board state to be copied into `own_board[]`. Since

```
THREADED queens( int n,                /* Size of board.          */
                 int row,              /* Current row.            */
                 int start_col,        /* Check from here to end. */
                 int *GLOBAL previous, /* Caller's board state.   */
                 int *GLOBAL result,   /* Where to send result.   */
                 SPTR done             /* Slot to sync when sent. */
                 )
{
  int col,
    own_board[MAX_BOARD_SIZE],
    sols_this_col,
    sols_other_cols;

  sols_this_col = 0;
  sols_other_cols = 0;
  BLKMOV_SYNC( previous, TO_GLOBAL(own_board), row*sizeof(int),
               DATA_RECEIVED );

  FIBER DATA_RECEIVED <* 1 *> {
    for( col = start_col; col < n; col++ ) {
      if ( safe(own_board, row, col) ) {
        own_board[row] = col;
        if ( row+1 == n ) {
          sols_this_col = 1;
          SYNC(DONE);
        } else {
          TOKEN( queens, n, row+1, 0, TO_GLOBAL(own_board),
                 TO_GLOBAL(&sols_this_col), TO_SPTR(DONE) );
        }
        if ( col+1 == n ) {
          SYNC(DONE);
        } else {
          TOKEN( queens, n, row, col+1, TO_GLOBAL(own_board),
                 TO_GLOBAL(&sols_other_cols), TO_SPTR(DONE) );
        }
        break;
      }
    }
    if ( col == n ) {
      SPAWN(DONE);
    }
  }

  FIBER DONE <* 2 *> {
    PUT_SYNC( sols_this_col + sols_other_cols, result, done );
    TERMINATE;
  }
}
```

Figure 13: Recursive N-Queens Code

the second arg of the BLKMOV_SYNC instruction (the destination into which the data is to be moved) is local, this block move is thus similar to the get-sync command discussed in the previous section (which copy remote data to local memory in a split-phase transaction), except that a more complex datum is moved. In this case, we are syncing a local sync slot (DATA_RECEIVED), where the last argument specifies a local slot identifier. Block moves can also be used like the data-syncs introduced in Section 4.5.1,

```
#define MAX_BOARD_SIZE 24

int safe( int board[], int row, int col )
{
  int rowchk, colchk;

  for( rowchk = 0; rowchk < row; rowchk++ ) {
    colchk = board[rowchk];
    if ( (col == colchk) ||
         (row - rowchk == col - colchk) ||
         (row - rowchk == colchk - col) )
      return( 0 );
  }
  return( 1 );
}

void print_usage_and_die()
{
  fprintf( stderr, "usage (with 1 <= size <= %d):\n    queens size\n",
           MAX_BOARD_SIZE );
  exit(1);
}

THREADED MAIN( int argc, char *argv[] )
{
  int n, result, place[MAX_BOARD_SIZE];

  if (argc != 2)
    print_usage_and_die();

  n = atoi(argv[1]);
  if ( n < 1 || n > MAX_BOARD_SIZE )
    print_usage_and_die();

  INVOKE( 0, queens, n, 0, 0, TO_GLOBAL(place), TO_GLOBAL(&result), TO_SPTR(DONE) );

  FIBER DONE <* 1 *> {
    printf( "Number of solutions for %d queens = %d\n", n, result );
    TERMINATE;
  }
}
```

Figure 14: N-Queens Checking Function and `MAIN` Routine

sending local data to a remote destination and synchronizing a slot at the destination end.[18]

[18]Actually, block moves are quite general. Any arbitrary combination of local and remote addresses is possible, 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 location 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 put-sync and get-sync flavors.

5 Advanced Features of Threaded-C

In this section we discuss more advanced features of Threaded-C, including the explicit declaration and binding of sync slots, the general form of fiber definitions, mutually exclusive fibers and atomic mailboxes, indexed fibers, global-scope variables, and polling.

5.1 Explicit Declaration and Binding of Sync Slots

In section 4, we saw the basic way in which a fiber can be defined and automatically associated with an appropriate sync slot. We will now examine in more detail the various ways — implicitly vs. explicitly — in which a slot can be declared, initialized, and bound to a fiber.

The simplest and certainly the most common way of defining a sync slot is by using the implicit slot associated with a fiber's definition. In this case, the slot's name is the same as that of the fiber and the initialization and reset counts are specified using the "`<* ... *>`" notation attached to the fiber, for example: `FIBER FOO <* init, rst *> { ... }`.

As mentioned previously, such a declaration introduces a sync slot named `FOO` associated with the fiber of the same name and is thus equivalent to the following explicit definition of a slot using the `INIT_SLOT` statements:

```
INIT_SLOT( FOO, init, rst, FOO );
...
FIBER FOO { ... }
```

Note that overloaded and abreviated forms of `INIT_SLOT` are also allowed. For example, all the following explicit `INIT_SLOT` instructions are equivalent:

```
INIT_SLOT( FOO, k, k, FOO );
INIT_SLOT( FOO, k, k );        /* fiber name  = slot name = FOO */
INIT_SLOT( FOO, k );           /* fiber name  = slot name = FOO
                                  reset count = init count      */
```

More generally, an `INIT_SLOT` instruction has the the following form:

```
INIT_SLOT( SLOT_NAME, init_count, reset_count, FIBER_NAME );
```

Note that this general form makes it possible to have multiple slots bound to the same fiber. For example, the following two instructions would imply that fiber `FOO` will be activated each time any one of slot `SLOT1` or `SLOT2` drops to 0:

```
INIT_SLOT( SLOT1, init1, rst1, FOO );

INIT_SLOT( SLOT2, init2, rst2, FOO );
```

Finally, it is also legal to have two specifications of the same slot, be they implicit and/or explicit. In turn, this also means that the fiber bound to a sync slot can be changed dynamically — `INIT_SLOT` is *not* a compile-time instruction. Note also that the implicit sync slots specifications are assumed to occur as the first statements of the threaded function, that is, before any of the other executable statements of the initialization fiber. Thus, the two pieces of code presented in Figure 15 would thus be considered equivalent, where the top part of the figure uses an implicit slot specification whereas the lower part gives the equivalent explicit instruction.

```
/* Implicit slot declaration and initialization. */
THREADED foo( ... )
{
  ... some declarations ...

  ... First instruction of foo() ...
  ...
  INIT_SLOT( F1, k1, k2 );        /* Explicit declaration of slot F1. */
  ...
  FIBER F1 <* k3 *> { ... }       /* Implicit declaration of F1.      */
  ...
}

/* Equivalent explicit slot definition and initialization. */
THREADED foo( ... )
{
  ... some declarations ...

  INIT_SLOT( F1, k3, k3, F1 );     /* Equivalent instruction. */

  ... First instruction of foo() ...
  ...
  INIT_SLOT( F1, k1, k2, F1 );
  ...
  FIBER F1 { ... }
  ...
}
```

Figure 15: Implicit vs. Explicit Sync Slot Initialization

5.2 General Fibers

In all examples presented so far, a fiber was defined by a strict textual sequence of code (block). However, in general, a fiber can be associated with (almost) any dynamic sequence of C code. In other words, a fiber can be interpreted as a form of *entry point*.

Two forms of these special fibers are particularly interesting:

1. Loops with split-phase operations: Suppose we have a loop where some remote data has to be obtained or produced in an iteration and then used later within that same iteration or in a subsequent iteration. In order to hide the long latency operation, the loop should terminate the current fiber's execution and resume only when the data becomes available. Figure 16 presents a simple example illustrating such a loop, where we want to print, one after the other, the result produced by function `fib` for different values of `n`.

   Each iteration of the main program's loop makes a call to the `fib()` function (Line 11). After each invocation in the loop, the fiber making the invocation is terminated when control flow reaches the `FIBER` keyword of fiber `NEXT` (Line 13), in order to wait for the result to arrive. When that result becomes available, fiber `NEXT`, which become enabled after receiving a single synchronization signal (Line 13), is activated and the received result is printed (Line 14). At this point, the normal flow of control simply goes back to the beginning of the loop, where another argument `n` is read and

```
1  THREADED MAIN( int argc, char *argv[] )
2  {
3    int k, i;
4
5    k = atoi(argv[1]);        /* How many different calls of fib to do. */
6
7    for( i = 0; i < k; i++ ) {
8      int n, res;
9
10     n = atoi(argv[i+2]);    /* Argument for the next call of fib. */
11     TOKEN( fib, n, TO_GLOBAL(&res), TO_SPTR(NEXT) );
12
13     FIBER NEXT <* 1 *>
14       printf( "fib(%d) = %d\n", n, res );
15   }
16   TERMINATE;
17 }
```

Figure 16: A Loop with a Split-phase Operation

another call to `fib()` is performed.

Note that this example does not imply that fibers are preemptive [19]; rather, it means that a fiber should not necessarily be interpreted as a pure straightline piece of code, and that a specific segment of code (e.g., Lines 7–11) can be part of two distinct fibers. Thus, in the above example, there are in fact two distinct fibers:[20]

(a) The first fiber initializes the loop and performs the first function call. This fiber is executed only once.

(b) The second fiber, which is executed `k-1` times, prints the result from the previous call to `fib()` and either goes on to perform the next iteration of the loop or exits the loop and terminates the function activation.

2. Conditional split-phase operations: As the name suggests, a conditional split-phase operation is one which may or may not be executed depending on some program condition. A typical example might be a fiber which is to be created only when it is recognized that some long latency operation needs to be performed. Figure 17 presents an example that implements some form of software cache: If the item is in the cache, no fiber switching will occur, because the `if` statement will simply not be entered. On the other hand, if the item is not in the cache, then the fiber's execution will be stopped after the `BLKMOV_SYNC`, when the `FIBER` is encountered. Fiber `NOW_IN_CACHE` will then be activated only when the split-phase access has completed, ensuring that the item is now in the cache.

What this last example illustrates is that although EARTH fibers are said to be non-preemptive, this should not be interpreted as meaning that fibers are simple straight line code blocks. Rather, it should be interpreted as meaning that there can never be a fiber switching not apparent in the executed code, that is, a fiber's execution can be terminated only because an explicit or implicit `END_FIBER` instruction [21] or

[19] It is important to stress that execution of a fiber is non-preemptive only in the sense that its execution can never be preempted by the execution of another program fiber (EU processing). However, this does not mean the execution of a fiber cannot be temporarily preempted by a lower-level RTS process or interrupt handler (SU processing).

[20] It is interesting to note that these two fibers can be seen as distinct stages of a two stages software pipeline, the second fiber representing the steady state stage.

[21] Recall that an implicit `END_FIBER` occurs before every `FIBER` keyword and before every threaded function closing bracket.

```
...
is_in_cache = ... /* Determine if item is present in cache. */
if ( !is_in_cache ) {
    BLKMOV_SYNC( ..., NOW_IN_CACHE );
    FIBER NOW_IN_CACHE <* 1 *>
      is_in_cache = TRUE;
}
/* The item is in the cache! */
...
```

Figure 17: An Example of a Conditional Split-phase Operation

a `TERMINATE` instruction has been executed. As for the `FIBER` keyword, it should be seen as introducing a kind of *entry point* into the flow of control (similar to a goto target address), but where synchronization signals and spawn instructions are the only mechanisms that can be used to trigger the execution of these entry points.

5.3 Indexed Fibers

Important Note: Due to time constraints, this feature may not yet available in the August 2000 release.

Suppose one wants to solve a problem by decomposing it into a fixed number of subproblems, let's say `N`, and that for each such subproblem a piece of code defined by a fiber `F` has to be performed. Using Threaded-C as defined so far, there is no easy way to do this without duplicating code. However, using *indexed* fibers, this can easily be done. The general form of a set of indexed fibers is the following:

```
FIBER FIB[i: 0..N-1] <* init, rst *> {
  /* Block of instructions which may (but need not) use i. */
  ...
}
```

This piece of Threaded-C code defines `N` distinct fibers, any of which can be activated and executed independently of the others, and each with its associated implicit sync slot noted `FIB[i]` (for `0 <= i < N`).

Note that if local variables are declared in a block of instructions associated with some indexed fibers, then each instance with distinct index will have independent copies of these local variables. Thus, no non-determinate behavior can occur in the following piece of code because of the non-atomic update of `x`:

```
SYNC( FIB[0] );
SYNC( FIB[1] );

FIBER FIB[i: 0..1] <* 1 *> {
  int x = init[i];
  x +=  a[i];
  printf( "x = %d\n", x );
}
```

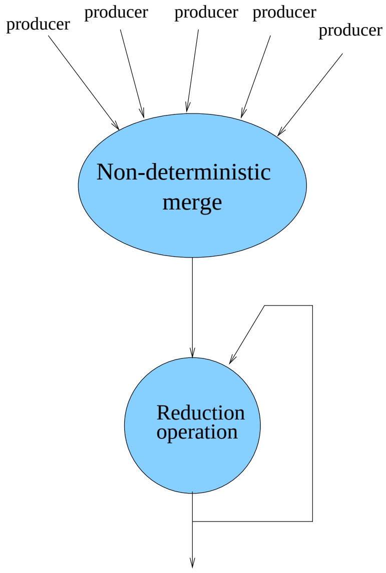

Figure 18: A Non-deterministic Merge Connecting Multiple Producers with a Unique Consumer

5.4 Mutually Exclusive Fibers and Atomic Mailboxes

Many Threaded-C programs can generally rely exclusively on strict data flow style synchronization to perform their task. However, there is an important feature of dataflow languages which cannot easily be expressed using the Threaded-C features introduced so far: the non-deterministic merge, which combines data from different sources into a single stream of data.

Suppose we have a number of producers that each generate a value and suppose that all these values have to be received by a unique consumer which then applies some reduction operation to these various incoming values. In a pure dataflow setting, this could be modeled by some form of non-deterministic merge where the outputs of all the producers would be connected to an appropriate merge operator; the output of this merge would then be used as the consumer's input, as illustrated in Figure 18.

In Threaded-C as presented so far, a value can be sent to some consuming fiber only if the producer has an explicit global handle to the location where the value will be stored. This situation is illustrated in Figure 19, where the consumer explicitly sends the destination address to the producer (Line 13), addess which is then used by the producer to send the produced value (Line 4), thus activating the appropriate fiber (Line 15).

The communication mechanism described above works well when there is a *unique* producer. However, such a simple strategy cannot work when there are multiple independent producers, because nothing would prevent a producer from overwriting the target location before fiber `X_READY` retrieves and processes the previously received value. In other words, since we have a unique location shared by multiple producers, this clearly is a case where some form of critical section would be needed in order to safely use this shared resource. Threaded-C supports two different mechanisms for mutual exclusion and critical sections: mutually exclusive fibers and atomic mailboxes.

Mutually exclusive fibers are introduced using the `EXCLUSIVE` keyword. These fibers are implemented in such a way that no two exclusive fibers from the same function activation can be executing concurrently at the same time. Atomic mailboxes are introduced by `MAILBOX` type declarations. Mailboxes allow producers, using various drop-in operations, to send data to a (possibly) remote data structure associated

```
THREADED producer( int *GLOBAL x, SPTR x_ready )
{
  int y = ...; /* Generate some value. */
  PUT_SYNC( y, x, x_ready );
  TERMINATE;
}

THREADED consumer( ... )
{
  int x;

  /* Activate producer, sending it x's target address. */
  TOKEN( producer, TO_GLOBAL(&x), TO_SPTR(X_READY) );

  FIBER X_READY <* 1 *> {
     /* Do something with x. */
     ...
  }
}
```

Figure 19: A Producer that Sends to a Unique Consumer

```
THREADED producer( MAILBOX *GLOBAL mb )
{
  int n = NODE_ID;

  DROP_IN( mb, &n, sizeof(int) );
  TERMINATE;
}

THREADED MAIN()
{
  MAILBOX mb;
  int i;
  int total = 0;

  INIT_MAILBOX( &mb, PROCESS_ITEM );
  for( i = 0; i < NUM_NODES; i++ )
    INVOKE( i, producer, TO_GLOBAL(&mb) );


  EXCLUSIVE FIBER PROCESS_ITEM <* 1 *> {
    int v;

    RETRIEVE_ITEM( mb, &v );
    total += v;
    SYNC(DONE);
  }

  FIBER DONE <* NUM_NODES *> {
    printf( "total = %d\n", total );
    TERMINATE;
  }
}
```

Figure 20: A Mailbox Example with a Mutually Exclusive Fiber

with the mailbox (bag-like, i.e., multiset, thus with no specific order) and such that, upon arrival of a new item, some special fiber (specified when the mailbox is created) will be informed of a new item having arrived, at which point the item can be retrieved. Use of these constructs is illustrated in Figure 20.

In addition to their initialization operation (`INIT_MAILBOX`), atomic mailboxes, which are introduced by the data type `MAILBOX`, support two major operations, as shown in Figure 20:

1. Items can be sent to a target mailbox, possibly from a remote location. Sending an item to a mailbox is also called "dropping" an item into the mailbox (`DROP_IN`), as shown in Line 5, where the value generated by the producer (Line 3: the `NODE_ID` on which the threaded function is being executed) is sent to the (remote) mailbox received as argument (Line 1).

   It is important to note that when an item is dropped into a mailbox, there is an implicit signal, specific to the mailbox instance, which is also sent. More precisely, when the item arrives at the target mailbox, a signal is sent to the sync slot specified when the mailbox was initialized (Line 15: `INIT_MAILBOX(...)`). Thus, in this example, sync slot/fiber `PROCESS_ITEM` (Line 20) will be signaled every time an item is dropped and arrives at the node owning mailbox `mb`.

2. Items can be retrieved from the mailbox (Line 23: `RETRIEVE_ITEM`). Contrary to the dropping operation which can be performed remotely, as in a regular (*snail* mail) mailbox items can only be retrieved locally.[22]

Other operations associated with atomic mailboxes are the following:

`void DROP_IN_SYNC( MAILBOX *GLOBAL mb, void *GLOBAL source,`
`int size, `*`SLOT_ID`*` source_free );`

In this variant, the `source` must be a global handle — not a local pointer contrary to `DROP_IN` — and the last argument indicates a sync slot which is to be signaled when the `source` location containing the item to be dropped into the mailbox can be safely overwritten without any effect on the value received by the target mailbox. Note that the `source_free` sync slot can be either a local (as indicated) or remote slot (`SPTR`).

`int RETRIEVE_ITEM_ADDR( MAILBOX mb, void **dest );`

This is similar to `RETRIEVE_ITEM` except that it simply copies into `*dest` the address of a buffer containing the retrieved item, not the item itself.

The various operations related with mailboxes are described in more detail in Appendix A.5.

These mailboxes are called *atomic* mailboxes because it is the responsibility of the compiler and run-time system to ensure that a correct behavior occurs if multiple items are being added and/or removed concurrently by different fibers. Atomicity, in our example, ensures that all the producers can safely perform their `DROP_IN` operation concurrently (Line 5) while at the same time the consumer tries to retrieve newly arrived item (Line 23).

Atomicity, in our example, also means that since fiber `PROCESS_ITEM`'s initial and reset counts are both 1 (Line 20), any item being dropped in the mailbox is sure to be retrieved from it, since the fiber will be synced every time an item is dropped.[23] However, atomicity of the mailbox operations *is not*

[22] Note that, in practice, this does not necessarily mean from the same function activation in which the mailbox was created. Rather, it means from a function on the *same node* as the mailbox.

[23] Although this is not used in this example, `RETRIEVE_ITEM` returns the size (number of bytes) of the item which was retrieved from the mailbox. When the returned value is 0, this means the mailbox was empty at the time the operation was performed.

sufficient to ensure that the program of Figure 20 behaves correctly, because the different instances of fiber PROCESS_ITEM — note that this is our first example where a specific fiber will definitively be activated more than once — all share a common resource, the integer variable `total`. Access to and update of this shared variable (Line 24) must also be protected in a critical section, in order to ensure that the updates are done atomically and that the right total is obtained. Here, the correct behavior is obtained simply by annotating fiber PROCESS_ITEM as being mutually EXCLUSIVE (Line 20).

A fiber annotated as being EXCLUSIVE means that, when it is being executed, it will always be the only fiber of its function activation being executed. More precisely, two EXCLUSIVE fibers from the same threaded function activation will never be allowed to execute concurrently. However, a non-EXCLUSIVE fiber could still be running at the same time as an EXCLUSIVE one.[24]

5.5 Memory Allocation

Dynamic memory allocation in Threaded-C is done using the standard operations (viz., `malloc`, `calloc`, `realloc` and `free`).[25] Each node has its own independent heap and, in the case of an SMP implementation, all processors on a given node share the same heap. Any memory management operation done using these operations is always performed through a local pointer, thus acts upon the local heap. Of course, Threaded-C functions can be defined in order to do some remote heap manipulation, in which case global handles and their associated operations have to be used.

As an illustration of how to perform both local and remote heap manipulation, we present a small number of threaded functions that allow to do this (Figure 21).

Line 1: Include the standard `assert` macro, which allows the verification of run-time assertions (see Line 17).

Line 3: Introduce an abreviation type to denote global handle.

Lines 7–13: Function `local_alloc()` allocates some space on the local heap (Line 11) and then returns the resulting global handle (Line 12) into the remote location passed as argument (`gh` at Line 7).

Lines 15–21: Releases some local heap memory. The first argument is a global handle; however, it must necessarily refer to a local address. This is first checked by the assertion in Line 17 — such an assertion aborts program execution if the indicated condition is false, in our case, if the executing node is not the owner of global handle `gh`.[26]

Lines 25–28: Used to allocate some heap memory on a remote node (first argument). The second argument (`gh`) must be a reference to a global handle and indicates where the resulting pointer will be stored.

[24]Note that some implementations of the RTS may allow multiple instances of the *same* fiber to execute concurrently, whereas some others may not. In the former case, the EXCLUSIVE annotation clearly ensures that two or more instances of the same exclusive fiber will *never* be executing concurrently. Note also that if the implementation allows multiple instances of the same (non-exclusive) fiber to execute concurrently, then whether the local variables of the fiber are shared between the different instances or whether distinct copies of these local variables are created is implementation dependent. Thus, a fiber containing local variables which may have to be executed multiple times should generally be annotated as being EXCLUSIVE.

[25]Note that the Threaded-C compiler and run-time system work under the assumption that these C library operations are *thread-safe*. The examples presented below also make the same assumption.

[26]This condition is equivalent to the following: `OWNER_OF(gh) == NODE_ID`.

```
1  #include "assert.h"
2
3  typedef  void *GLOBAL GlobalHandle;
4
5  /* Local operations. */
6
7  void local_alloc( GlobalHandle *GLOBAL gh, int size, SPTR done )
8  {
9    void* ptr;
10
11   ptr = malloc(size);
12   PUT_SYNC( TO_GLOBAL(ptr), gh, done );
13 }
14
15 void local_free( GlobalHandle gh, SPTR done )
16 {
17   assert( IS_LOCAL(gh) );
18
19   free( TO_LOCAL(gh) );
20   SYNC( done );
21 }
22
23 /* Remote operations. */
24
25 void remote_alloc( int node, GlobalHandle *GLOBAL gh, int size, SPTR done )
26 {
27   INVOKE( node, local_alloc, gh, size, done )
28 }
29
30 void remote_free( GlobalHandle *gh, SPTR done )
31 {
32   INVOKE( OWNER_OF(*gh), local_free, *gh, done );
33   *gh = MAKE_GPTR( (void*) NULL, 0 );
34 }
35
```

Figure 21: Allocate and Remote Deallocate without Global-Scope Variables

Lines 30–34: `remote_free()` simply determines, via `OWNER_OF` in Line 32, the node where the appropriate `local_free` operation must be performed. To clearly indicate that the global handle passed as (a reference) argument is not valid after the call, we set it to a `NULL` global handle (Line 33).

Figure 22 shows code that uses `remote_alloc()` and `remote_free()`. In Line 6 we (locally) allocate an array of global handles (in node 0). Its entries refer to arrays which we dynamically allocate in other nodes by calling `remote_alloc()` (Lines 8–9: we assume that function `size_for()` returns the size required for the array to be allocated on node `i`). Finally, these entries are deallocated, in Lines 11–13, through the calls made to `remote_free()`. Note that although each of these calls to `remote_free()` (and similarly for `remote_alloc()`) is done synchronously using a regular function call, each of them does create a parallel invocation of `local_free()` on the appropriate node (Line 32 of Figure 21); thus, this loop is in fact a for-all loop, with fiber `DONE` (Line 13 of Figure 22) playing the role of the barrier.

```
THREADED MAIN( int argc, char* argv[] )
{
  int          i;
  GlobalHandle* table;

  table = (GlobalHandle*) malloc( NUM_NODES*sizeof(GlobalHandle) );

  for( i = 0; i < NUM_NODES; i++ )
    remote_alloc( i, TO_GLOBAL(&table[i]), size_for(i), TO_SPTR(DO_DEALLOC) );

  FIBER DO_DEALLOC <* NUM_NODES *>
    for( i = 0; i < NUM_NODES; i++ )
      remote_free( &table[i], TO_SPTR(DONE) );

  FIBER DONE <* NUM_NODES *>
    free(table);
    TERMINATE;
}
```

Figure 22: `MAIN` Function Calling Global-Free `local_alloc` and `remote_free`

5.6 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 (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. 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.

Note that `POLL` commands can be used inside of both threaded and sequential functions.

6 Examples of Threaded-C Programming

In this section, we illustrate the language constructs covered in the previous two sections with some larger, more complex code examples. The first example shows how split-phase locks, a typical mechanism to enforce mutual exclusion in shared-memory style programming, can be implemented using exclusive

```
#ifndef LOCK_H
#define LOCK_H

/* Public operations. */

typedef struct LOCK *GLOBAL LOCK;

void INIT_LOCK( LOCK *lock, SPTR created );
void LOCK_SYNC( LOCK lock, SPTR slot );
void UNLOCK   ( LOCK lock );
void FREE_LOCK( LOCK lock );


/* Private stuff. */

THREADED _INIT_LOCK( LOCK *lock, SPTR created );
THREADED _LOCK_SYNC( LOCK lock, SPTR slot );
THREADED _UNLOCK   ( LOCK lock );
THREADED _FREE_LOCK( LOCK lock );


#define INIT_LOCK(l, s) INVOKE(NODE_ID,       _INIT_LOCK, (l), (s))
#define LOCK_SYNC(l, s) INVOKE(OWNER_OF((l)), _LOCK_SYNC, (l), (s))
#define UNLOCK(l)       INVOKE(OWNER_OF((l)), _UNLOCK,    (l))
#define FREE_LOCK(l)    INVOKE(OWNER_OF((l)), _FREE_LOCK, (l))

#endif
```

Figure 23: Interface (Header File) for a Module Defining Split-phase Locks

fibers and atomic mailboxes. The second example shows one way to implement a one-dimensional FFT (Fast Fourier Transform).

6.1 Example: Implementation of split-phase locks

Our first example will illustrate how atomic mailboxes and exclusive fibers can be used to implement synchronization mechanisms often defined as primitive operations in other parallel programming languages. Here, we will show how locks, a typical mechanism used to enforce mutual exclusion when using shared-memory style parallel programming, can be realized at the Threaded-C programming level.

First, recall that locks are used to implement mutual exclusion by using the following strategy: before a critical section can be entered by a process, this process first has to obtain ownership of an appropriate lock; once the lock has been acquired, the process can perform the critical section's work, after which it releases the lock. Mutual exclusion is enforced by ensuring that any other process which tries to grab ownership of the lock when it is already in use (locked) gets suspended until the lock is released. Of course, this also means the locking operation must be performed atomically and, unless busy waiting is used, the identity of the waiting processes must be kept in some appropriate data structure, so that when the owning process releases the lock, one of the waiting processes can be granted ownership.

Now, given the multi-threaded nature of Threaded-C and the non-preemptive nature of fibers, it would clearly not be acceptable for a fiber to spin-wait for a lock to become available. Instead, obtaining a lock should be done, much like a read/fetch operation, in a split-phase manner. Figure 23 presents the interface (C header file) for a module that define such split-phase locks:

Lines 1–2: Typical header with definition of symbolic label unique to this file to ensure that no problem occurs if the header file is `include`d multiple times, either directly or indirectly.

Lines 6–11: We declare the new type which is to be exported by this module, along with related operations. Note that a lock is a global handle to some (private) structure. Note also that, as we will see later, the public operations can be called *without* using explicit `INVOKE` statements, which is why the operations have all been declared of type `void`. For example, to release a lock `l`, one will simply need to write "`UNLOCK(l);`".

Lines 16–19: These declarations define the private names for the operations which will effectively be invoked to manipulate locks: a lock being always refered through a global handle, an owner-based approach will be used, where the real work will be done on the processor owning the lock.

Lines 22–25: These macros redefine the publicly exported operations (defined in Lines 8–11) and make it possible to somewhat hide the owner-based invocation that must be performed when applying an operation to a (possibly remote) lock, thus making it simpler to manipulate locks. It is because of these macros that a call such as "`UNLOCK( l );`" will be possible and correct, since this will automatically be mapped into "`INVOKE(OWNER_OF(l), _UNLOCK, l)`".

The intuitive idea behind our implementation of split-phase locks using atomic mailboxes is to use the mailbox as the data structure that will contain the identity of the processes which are waiting for ownership of the lock. Here, this means the mailbox will contain the remote slots (`SPTR`) which have to be signaled when ownership of the lock is granted, slots which are specified when the `LOCK_SYNC` operation is performed (Line 9 of Figure 23). Thus, performing a lock operation will only require dropping an appropriate `SPTR` value into the mailbox associated with the lock (see Figure 25, which is explained in more detail below).

Figure 24 presents the key function for implementing the locks. For each independent lock, a threaded function invocation that plays the role of a *lock handler* (Line 10) is created. It is then fibers inside this lock handler which are responsible to grant ownership of the lock to the fibers that ask for the lock and then later release it. The identity of the requesting fibers (more precisely, the sync slot to be signaled in a split-phase fashion) will have been dropped in the mailbox associated with the lock (Line 4 of the `MAILBOX` data structure definition) and will later be retrieved (Line 28) by fiber `LOCK_ASKED`. In more detail, here is how the lock handler works:

Lines 3–8: A `LOCK` structure is described by an atomic mailbox, a fiber within the associated lock handler that must be signaled when an unlock is performed, a fiber to signal when the lock has to be freed, and a variable that counts the number of unlocks which are still pending (for which a locking operation has been demanded but the corresponding unlocking has not yet been done, either because the lock has not yet been obtained or because the critical section is not yet fully executed).

Lines 19–23: The lock structure is initialized. Note (Line 22) that the mailbox's signal upon receiving an item will be sent to fiber `LOCK_ASKED` of function `lock_handler` (see next item).

Lines 25–34: This is the fiber that is signaled everytime an item is dropped in the associated mailbox, thus every time a call is made to `LOCK_SYNC`. If no unlock is pending (condition in Line 26 is true), then this means the lock is free and can be granted immediately; otherwise (Lines 30–31), we have to wait for an unlock to be performed and nothing needs to be done for now, as the identity of the lock's new demander is already in the mailbox.

```
1  #include "LOCK.h"
2
3  struct LOCK {
4    MAILBOX mailbox;
5    SPTR    unlock;
6    SPTR    free_lock;
7    int     nb_pending_unlocks;
8  };
9
10 THREADED lock_handler( LOCK lock_handle, SPTR created )
11 {
12   struct LOCK* lock;
13   SPTR         slot_to_sync;
14   int          size;
15
16   assert( IS_LOCAL(lock_handle) );
17   lock = TO_LOCAL(lock_handle);
18
19   lock->nb_pending_unlocks  = 0;
20   lock->unlock              = TO_SPTR(UNLOCK);
21   lock->free_lock           = TO_SPTR(FREE_LOCK);
22   INIT_MAILBOX( &(lock->mailbox), LOCK_ASKED );
23   SYNC(created);
24
25   EXCLUSIVE FIBER LOCK_ASKED <* 1 *> {
26     if ( lock->nb_pending_unlocks == 0 ) {
27       /* No owner => give it away immediately. */
28       RETRIEVE_ITEM( lock->mailbox, &slot_to_sync );
29       SYNC( slot_to_sync );
30     } else {
31       /* Already owned: do nothing and leave demander into mailbox. */
32     }
33     lock->nb_pending_unlocks++;
34   }
35
36   EXCLUSIVE FIBER UNLOCK <* 1 *> {
37     lock->nb_pending_unlocks--;
38     if ( lock->nb_pending_unlocks > 0 ) {
39       /* Some other process is waiting for it. */
40       RETRIEVE_ITEM( lock->mailbox, &slot_to_sync );
41       SYNC( slot_to_sync );
42     }
43   }
44
45   FIBER FREE_LOCK <* 1 *> {
46     FREE_MAILBOX( lock->mailbox );
47     TERMINATE;
48   }
49 }
```

Figure 24: Implementation of Split-phase Locks: The `lock_handler` Threaded Function

Lines 36–43: This fibers gets called when an unlock has to be performed. If there are demanders waiting to obtain the lock (condition in Line 38 is true), then one of them is retrieved from the mailbox and a signal is sent to indicate it now has ownership of the lock (Line 41).[27]

[27] To simplify the presentation, we have omitted code to check that `nb_pending_unlocks` does not go below 0, in cases where more unlocking than locking has been done.

```
50 THREADED _INIT_LOCK( LOCK *lock_handle, SPTR created )
51 {
52   struct LOCK *lock;
53
54   lock = (struct LOCK *) malloc( sizeof(struct LOCK) );
55   *lock_handle = TO_GLOBAL(lock);
56   INVOKE( NODE_ID, lock_handler, *lock_handle, created );
57   TERMINATE;
58 }
59
60 THREADED _LOCK_SYNC( LOCK lock, SPTR slot )
61 {
62   DROP_IN( TO_GLOBAL(&TO_LOCAL(lock)->mailbox), &slot, sizeof(SPTR) );
63   TERMINATE;
64 }
65
66 THREADED _FREE_LOCK( LOCK lock )
67 {
68   SYNC( TO_LOCAL(lock)->free_lock );
69   TERMINATE;
70 }
71
72 THREADED _UNLOCK( LOCK lock )
73 {
74   SYNC( TO_LOCAL(lock)->unlock );
75   TERMINATE;
76 }
```

Figure 25: Implementation of Split-phase Locks Using Atomic Mailboxes

Lines 45–48: The lock has to be freed, so we also reclaim the associated mailbox space and then terminate the lock handler's execution.

Given the lock handler described in Figure 24, it is then quite straightforward to implement the various (private) lock operations, as shown in Figure 25. Note in Line 56 the explicit creation of the lock handler associated with a newly created lock. As for the other operations, since they are always executed on the node owning the lock and since all the real work is done by the associated lock handler, note how simple they are: they just send a sync signal, either implicitly (Line 62) or explicitly (Lines 68 and 74), to an appropriate sync slot within the handler.

These split-phase locks can be used, as presented in Figure 26, to implement a program similar to the one written using a mailbox that was introduced in section 5.4 (Figure 20), where a number of producers send some value to a consumer that computes their sum. The first step in the producer (Line 5) consists in obtaining the lock that will ensure that the update, performed by fiber `PROCESS_ITEM` (Line 26), is done atomically. Note that the unlocking is done by the consumer (Line 29). Note also that the fiber performing the update does not need to be annotated as `EXCLUSIVE` since it is protected by the lock acquired in the producer, so that multiple instances of that fiber will never be executing concurrently.

This solution to the multiple producers/single consumer is more costly, in terms of communication operations, than the one introduced in Section 20. In the solution presented in Figure 20, a single communication is required to send data from the producer to the mailbox; in the solution of Figure 26, each data transfer requires 3 communications: one to request the lock, one to receive ownership of the lock, and one to send the value.

```
THREADED producer( LOCK l, int *GLOBAL v, SPTR item_ready )
{
  int n = NODE_ID;

  LOCK_SYNC( l, TO_SPTR(LOCKED) );

  FIBER LOCKED <* 1 *> {
    PUT_SYNC( n, v, item_ready );
    TERMINATE;
  }
}

THREADED MAIN()
{
  LOCK l;
  int i, v;
  int total = 0;

  INIT_LOCK( &l, TO_SPTR(INVOKE_PRODUCERS) );

  FIBER INVOKE_PRODUCERS <* 1 *> {
    for( i = 0; i < NUM_NODES; i++ )
      INVOKE( i, producer, l, TO_GLOBAL(&v), TO_SPTR(PROCESS_ITEM) );
  }

  FIBER PROCESS_ITEM <* 1 *> {
    total += v;
    UNLOCK(l);
    SYNC(DONE);
  }

  FIBER DONE <* NUM_NODES *> {
    printf( "total = %d\n", total );
    TERMINATE;
  }
}
```

Figure 26: Programs Using Split-phase Locks to Sum Values Generated by Independent Consumers

6.2 Example: One-Dimensional Fast Fourier Transform

We conclude this section with another example: the parallelization of a 1D FFT. The program below is a 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, as it is a naïve solution that neglects certain constraints of a real architecture. Implementations of FFT in Threaded-C having better performance are presented in [9].

Algorithmic Background: Consider a sequence

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

of 2^N complex numbers. Its Fourier transform

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

```
#ifndef Complex_H
#define Complex_H

typedef struct Complex{ double re; double im; } Complex;

Complex c_create( double re, double im );

Complex c_add ( Complex c1, Complex c2 );
Complex c_sub ( Complex c1, Complex c2 );
Complex c_mult( Complex c1, Complex c2 );

Complex c_scale( Complex c0, double scale );

Complex c_exp( Complex c0 );

#endif
```

Figure 27: Header File for Complex Number Operations

is given by

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

where

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

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 $j = 0, ..., 2^{N-1}-1$:

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

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 recursion. This example illustrates the use of `BLKMOV_SYNC` and shows an application of the `TOKEN` mechanism.

Note that we will also need appropriate operations for manipulating complex numbers. The header file for these operations is shown in Figure 27. The `MAIN` program does nothing but read the data and call the function `fft_token()` on the node where the initial data vector resides, so it will be omitted.

The Core Routine in Threaded-C: The main features of the core `fft_token()` routine are the followings:

```
1  THREADED fft_token( Complex *GLOBAL data, unsigned int n, int isign,
2                      SPTR done )
3  {
4    unsigned long   i, length, half_length;
5    Complex         *data_even, *data_odd, *w;
6    Complex         phase;
7
8    length = 1 << n;
9    half_length = length >> 1;
10
11   data_even = (Complex*) malloc(length*sizeof(Complex));
12   data_odd = data_even + half_length;
13
14   /* Fetch original data. */
15   BLKMOV_SYNC( data, TO_GLOBAL(data_even), length*sizeof(Complex),
16                DATA_READY );
17
18   /* Compute required roots of unity. */
19   w = (Complex*) malloc(half_length*sizeof(Complex));
20   phase = c_create(0.0, M_PI * ((double) isign) / ((double) half_length));
21   for( i = 0; i < half_length; i++ )
22     w[i] = c_exp( c_scale(phase, (double) i) );
23   SYNC( DATA_READY );
```

Line 1: The arguments of `fft_token()` consist of a global handle to the array to be transformed, the base 2 logarithm of its length, a sign (± 1) indicating whether we do a forward or backward Fourier transform, and a remote sync slot to signal completion.

Lines 8–23: Fetch (Lines 15–16) the array of complex numbers into a local array (allocated in Line 11, whose length is computed in Line 8) and overlap the retrieval with the computation of the necessary complex roots of unity (Lines 19–22). This overlapping is obtained by initiating the data transfer operation as soon the local array is allocated, after which the computation of the elements of array `w` (the complex roots of unity) is performed. Note that in order to be completely synchronization-safe, a sync signal from the initialization thread also has to be sent to fiber `DATA_READY` (Line 23) to indicate that both the computation of `w` is complete and the remote data has been copied locally.

Line 25–28: If recursive calls have to be made (bottom of recursion not yet reached), then the even part of the array has to be shuffled into the lower half (`data_even`) and the odd part into the upper half (pointed to by `data_odd`), after which the recursive calls can be made by generating two tokens to transform the even and the odd parts.

Line 30: This is the base case of the recursion: we can perform the appropriate operations (fiber `DO_OPERATIONS`) right away by spawning directly the associated fiber.

Lines 33–43: The various complex arithmetic operations can now be performed. Note that if `n = 1` (base case of recursion), then only a single iteration of the `for`-loop (Lines 34–39) will be required. After the loop is over, we send the result back to where we fetched the untransformed data (Line 40). This can be overlapped with freeing the array of complex roots (Line 41). Again an additional sync signal must be sent (Line 42).

Lines 45–49: The result has been sent to its proper destination and the array `w` has been freed. We can then send the proper termination signal to the caller (Line 46), free the space allocated for the local copy of the data and then terminate.

```
  FIBER DATA_READY <* 2 *> {
    if ( n > 1 ) {
      shuffle( data_even, n );
      TOKEN( fft_token, TO_GLOBAL(data_even), n-1, isign, TO_SPTR(DO_OPERATIONS) );
      TOKEN( fft_token, TO_GLOBAL(data_odd),  n-1, isign, TO_SPTR(DO_OPERATIONS) );
    } else {
      SPAWN( DO_OPERATIONS );
    }
  }

  FIBER DO_OPERATIONS <* 2 *> {
    for ( i = 0; i < half_length; i++ ) {
      Complex z_e = data_even[i];
      Complex z_o = data_odd[i];
      data_even[i] = c_add(z_e, c_mult(w[i], z_o));
      data_odd[i]  = c_sub(z_e, c_mult(w[i], z_o));
    }
    BLKMOV_SYNC( TO_GLOBAL(data_even), data, length*sizeof(Complex), DONE );
    free( w );
    SYNC( DONE );
  }

  FIBER DONE <* 2 *> {
    SYNC( done );
    free( data_even );
    TERMINATE;
  }
}
```

Figure 28: FFT Code

Of course, this program could be optimized in various ways. For example, we could precompute the complex roots of unity (array `w`) or throttle the recursion at a certain level and then continue sequentially.

7 Differences Between Threaded-C and ANSI C

The programmer should be aware that some standard C features are not fully supported in the current version of Threaded-C. This section lists the unsupported features.

- A threaded function cannot be static.
- A threaded function cannot have a variable number of arguments ("..." syntax).
- Although initializers should be allowed for both local and global variables, static or not, complex structured initial values may not yet be implemented. For example:

```
/* Currently allowed. */
int x = 0;
static int y = 0;
char* s = "abcdef";

/* May or may not be implemented. */
int str[] = {10, 20, 30, 40};
struct {int x; float y;} s = {10, 20.3};
```

8 Conclusion

This document has presented release 2.0 of the 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, use of atomic mailboxes, allocation of memory in remote nodes, all using the Threaded-C primitives.

Threaded-C is still 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, including some public libraries defining higher-level synchronizations mechanisms, is still currently under development. As we do so, we will post these improvements on the Web. Please regularly check the following URL: `http://www.capsl.udel.edu/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 [7]. 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. Nelson J. Amaral, Zachary Ruiz, Sean Ryan, Andrés Marquéz, and Prasad Kabulavarapu took part in the effort that resulted in release 1.1 of Threaded-C [1] which introduced the use of symbolic names for sync slots and fibers. The present document was developed partly based on a previous document [12] in the development of which Gerd Herber, Olivier Maquelin and Xinan Tang also participated, in addition to some of the authors of the present one (Kevin B. Theobald, J. Nelson Amaral and Guang R. Gao).

ACAPS and CAPSL members who have contributed to the EARTH runtime system include Haiying Cai, Prasad Kakulavarapu, Cheng Li, Andres Marquez, and Shashank Nemawarkar. 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, 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] José Nelson Amaral, Zachary Ruiz, Sean Ryan, Andres Marquez, Christopher Morrone, Prasad Kakulavarapu, and Guang R. Gao. Portable Threaded-C release 1.1. CAPSL Technical Note 05, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, October 1998.

[2] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Parallel Processing Symposium* [8], pages 704–709.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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* [8], pages 288–294.

[8] IEEE Computer Society. *Proceedings of the 8th International Parallel Processing Symposium*, Cancún, Mexico, April 26–29, 1994.

[9] Ashfaq A. Khokhar, Gerd Heber, Parimala Thulasiraman, and Guang R. Gao. Load adaptive algorithms and implementations for the 2D discrete wavelet transform on fine-grain multithreaded architectures. In *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing*, pages 458–462, San Juan, Puerto Rico, April 12–16, 1999. IEEE Computer Society and ACM SIGARCH.

[10] 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.

[11] 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.

[12] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[13] G. Tremblay. Threaded-C release 2.0: Motivation, description, and rationale. CAPSL Technical Note 09, University of Delaware, June 2000.

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 Fibers and Threaded Functions

`THREADED`

Keyword for introducing a threaded function declaration.

`FIBER` **fiber_name**

Marks the beginning of a fiber. This label must be unique within the threaded function. When the normal C flow of control reaches the `FIBER` keyword, then control switches to another ready fiber. The fiber label can also be followed by an init count, or by both an init and a reset count, using the `<* ... *>` notation.

`EXCLUSIVE FIBER fiber_name`

The `EXCLUSIVE` annotation indicates that a fiber should always be the only `EXCLUSIVE` fiber within its function activation to be executing at any given time. More precisely, two `EXCLUSIVE` fibers of the same function activation (whether they are two instances of the same fiber or instances of two different fibers) are never allowed to run at the same time. Note, however, that a non-`EXCLUSIVE` fiber can still be running at the same time as an `EXCLUSIVE` one. Also note that the initialization fiber is not exclusive.[28]

`void END_FIBER`

When an `END_FIBER` is executed, control switches to another ready fiber. In release 2.0 of Threaded-C, `END_FIBER` instructions rarely need to be used explicitly, because an implicit `END_FIBER` is assumed to appear just before every `FIBER` keyword and just before a threaded function closing bracket.

`int NUM_NODES`

Run-time system variable set to the number of nodes available in the system. Although the init and reset counts that follow a fiber label (see above) must be compile-time expressions, use of `NUM_NODES` is also allowed in these expressions.

Note that in the current SMP implementation, this returns the total number of processors (sometimes called virtual nodes).

`int NODE_ID`

Run-time system variable set to the identification number for the local node. This number always ranges from 0 to `NUM_NODES - 1`. Although the init and reset counts that follow a fiber must be compile-time expressions, use of `NODE_ID` is also allowed in these expressions.

Note that in the current SMP implementation, `NODE_ID` returns the processor id (sometimes called the virtual node id). This means that two different node ids may still identify nodes which do share some memory: see `SHARE_MEMORY()` below.

[28]This restriction is easy to get around by introducing a new explicitly named fiber and then simply spawning it in the real initialization fiber:

```
SPAWN( NEW_INIT_FIBER );
EXCLUSIVE NEW_INIT_FIBER <* 1 *> ...
```

void POLL

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 fibers can significantly improve overall performance because it allows for the faster handling of external requests.

void CALL(func_name, ...)

Calls the threaded function func_name and blocks until that function terminates. This is the mechanism to allow the execution of a threaded function without creating a parallel invocation of it (without doing a fork, that is, like a *synchronous* function call).

Note: In most implementations, when the CALL instruction gets executed, the initialization fiber of the called function (callee) gets executed right away. In other words, although the initialization fiber of the callee is a fiber distinct from the one where the CALL instruction occured (caller), execution of that initialization fiber does not incur any fiber switching.

Note: If an exclusive fiber contains a CALL instruction and the called function contains multiple fibers, then one should not expect the continuation code (after the return from the CALL) to still be part of the critical section. In other words, a CALL instruction in the middle of an exclusive fiber should *conceptually* be interpreted as follows:

```
CALL( foo, a1, ..., ak )

==

INVOKE( NODE_ID, foo, a1, ..., ak, TO_SPTR(CALL_DONE) );
FIBER CALL_DONE <* 1 *> {}
```

void INVOKE(int node, func_name, ...)

Creates a new parallel invocation of threaded function func_name on the specified node. Does not block nor wait for the invoked function to terminate; in others words, this is similar to a fork operation. The initialization fiber of the invoked function is put in the pool of ready fibers, similar to an implicit spawn of that initialization fiber.

To invoke a function foo() on all the processors, the following for-loop can be used, whether the implementation is an SMP one or not:

```
for( i = 0; i < NUM_NODES; i++ )
  INVOKE( i, foo, ... );
```

void TOKEN(func_name, ...)

Creates a new parallel invocation of threaded function func_name on a node selected by the runtime system. The node is selected trying to optimize the distribution of the workload in the machine. The initialization fiber of the invoked function is put in the pool of ready fibers on the selected node.

void TERMINATE

Terminates the execution of a threaded function, deallocating its associated frame.

A.2 Fiber Synchronization

Fibers are associated with sync slots. The current value (current count) associated with that slot represents how many signals the fiber has to receive before it can be activated. The programmer can initialize the sync count and update the count to control the firing of a fiber. We use the EARTH primitives described below to operate on sync slots and fibers.

First, we introduce the following two types which are used strictly to describe those primitives. Thus, these two types are not part of Threaded-C *per se* and cannot be used in programs.

SLOT_ID
: This type indicates an identifier associated with a local sync slot.

FIBER_ID
: Similar to *SLOT_ID*, but for fiber names.

`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 the synchronization slots using the pre-defined type `SLOT`. If used explicitly, this declaration must appear at the beginning of a function. However, threaded functions that use fibers with identifier labels do not need to declare this array.

`SPTR`
: A pre-defined type for synchronization pointers. It is defined as follows:

```
typedef SLOT *GLOBAL SPTR;
```

`SPTR SYNC_SLOTS_BASE( void )`
: Returns the base address of the array of sync slots for the current threaded function. Of course, this can be used even if the array has only been implicitly allocated, as occurs when symbolic slot and fiber names are used.

`int SLOT_OFFSET(` *SLOT_ID* `s )`
: Returns the offset, relative to the local sync slots array, of the local slot identified by `s`. This operation, combined with the previous one, allows a programmer to do various advanced manipulations of the sync slots array elements.

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

`void *IP_ADR(` *FIBER_ID* `f )`
: Returns a (local) pointer to the first instruction of fiber `f`. 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 TO_SPTR(` *SLOT_ID* `s )`
: Returns a global pointer to sync slot `s` (a local slot).

`void INIT_SLOT(` *SLOT_ID* `s, int count )`
: Initializes sync slot `s` with the initial counter value `count`, the reset value `count`, and a fiber pointer for the fiber with the same name as `s`.

`void INIT_SLOT(` *SLOT_ID* `s, int init_count, int reset_count )`
: Initializes sync slot `s` with the initial counter value `init_count`, the reset value `reset_count`, and a fiber pointer for the fiber with the same name as `s`.

`void INIT_SLOT( SLOT_ID s, int init_count, int reset_count, FIBER_ID f)`
: Initializes sync slot `s` with the initial counter value `init_count`, the reset value `reset_count`, and a fiber pointer for fiber `f`.

`void SYNC( SLOT_ID s )`
`void SYNC( SPTR s )`
: Decreases the sync count of slot `s` by one, which can either be a local or remote slot. If the count reaches zero the associated fiber is scheduled for execution.

`void INCR_SLOT( SLOT_ID s, int val )`
`void INCR_SLOT( SPTR s, int val )`
: Increases the sync count of slot `s` by `val`; `s` can either be a local or remote slot. If the count becomes zero the associated fiber is scheduled for execution.

`void SPAWN( FIBER_ID f )`
: Schedules local fiber `f` for execution.

`void SPAWN( void *GLOBAL fp, void *ip )`
: Schedules a remote fiber for execution. The fiber is specified through its frame (`fp`) and instruction (`ip`) pointers.

Implicit sync operation
: All data transfer primitives also perform a sync operation after the data has reached its destination. More generally, any instruction whose name terminates with `SYNC` performs such a synchronization.

Numeric sync slots
: [For advanced users only:] It is possible to mix symbolic fiber and slot names with explicit fiber and slot numbers. For example, the following program would be legal:

```
INIT_SLOT( SLOT1, 1, 1, 2 );
INIT_SLOT( 2, 1, 1, FIBER1 );
INIT_SLOT( 1, 1, 1, FIBER2 );
INIT_SLOT( DONE, 1, 1, DONE );
...
FIBER FIBER1 ...  // Associated with slot 2
FIBER 2      ...  // Associated with slot SLOT1 (= 0)
FIBER FIBER2 ...  // Associated with slot 1
FIBER DONE   ...  // Associated with slot DONE (= 3)
```

There are also cases where a programmer may want to know or specify the relative positions of specific slots. This can be done in two different ways:

1. By introducing explicit `define` statements for the required slots. For example, the following piece of code would ensure that the slots for fibers `FOO1` and `FOO2` are adjacents to each other:

   ```
   #define FOO1 0
   #define FOO2 1

   FIBER FOO1 <* i1, r1 *> { ... }

   FIBER FOO2 <* i2, r2 *> { ... }
   ```

2. By computing the difference between the associated offset. For example, the following value could be used to perform global handle pointer arithmetic:

   ```
   #define OFF12 (SLOT_OFFSET(FOO2) - SLOT_OFFSET(FOO1))
   ```

A.3 Data Transfer Primitives

The data transfer primitives support remote memory accesses and block data transfers. Transfers of values of any type which can be passed *by value* are supported by PUT_SYNC and GET_SYNC. Other transfers (e.g., for complete arrays) can be performed using BLKMOV_SYNC operations.

For all such data transfer operations, the sync slot that should be signaled when the operation terminates can either be specified as a (local) slot name, or as a global pointer (SPTR). Here are the basic communication primitives:

`void PUT_SYNC( T datum, T *GLOBAL dest, SLOT_ID s )`

`void PUT_SYNC( T datum, T *GLOBAL dest, SPTR s )`

Sends a value to the destination address and then update the specified sync slot.

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

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

Reads a value (of the appropriate type) from the source address and copies it to the destination address. Then, updates the specified sync slot.

`void BLKMOV_SYNC( void *GLOBAL src, void *GLOBAL dest, long length, SLOT_ID s )`

`void BLKMOV_SYNC( void *GLOBAL src, void *GLOBAL dest, long length, SPTR s )`

Copies `length` bytes of data from the source (`src`) to the destination (`dest`) address and then signals the specified sync slot when the data has reached its final destination.

```
void BLKMOV_SYNC( void *GLOBAL src, void *GLOBAL dest,
     long length, SLOT_ID source_free, SLOT_ID dest_ready )
```

```
void BLKMOV_SYNC( void *GLOBAL src, void *GLOBAL dest,
     long length, SLOT_ID source_free, SPTR dest_ready )
```

```
void BLKMOV_SYNC( void *GLOBAL src, void *GLOBAL dest,
     long length, SPTR source_free, SLOT_ID dest_ready )
```

```
void BLKMOV_SYNC( void *GLOBAL src, void *GLOBAL dest,
     long length, SPTR source_free, SPTR dest_ready )
```

Copies `length` bytes of data from the source (`src`) to the destination (`dest`) address. Signals `source_free` as soon as the source data can be safely overwritten. Signals `dest_ready` when the data has been copied to its final destination.

A.4 Global Address Support

`GLOBAL`

Type qualifier used to distinguish global handles from local (normal) pointers.

`T *GLOBAL TO_GLOBAL( T* ptr )`

Turns a local pointer into a global handle that refers to address `ptr` on the local node. In the portable implementation the type of the result depends on the type of the argument.[29]

[29] On MANNA, the result is of type pointer to `void`.

`T *TO_LOCAL( T *GLOBAL gptr )`
: From a global handle, obtains its local pointer component (extracts the address part of a global pointer). Note that it is *not* possible to dereference a global pointer without first turning it into a local pointer.[30]

`T *GLOBAL MAKE_GPTR( T *ptr, int node )`
: Takes a node number and a local address and returns the corresponding global handle.[31]

`int OWNER_OF( T *GLOBAL gptr )`
: Returns the node pointed to by `gptr` (extracts the node part of a global handle). In the SMP implementation, this returns the virtual node number.

`int IS_LOCAL( T *GLOBAL gptr )`
: Determines (*true* or *false*) if `gptr`'s associated address refers to a location in the node's local memory.

`int SHARE_MEMORY( int n1, int n2 )`
: Determines (*true* or *false*) if two nodes share the same memory module/space.

In summary, the following relationships among the above operations will hold:

```
OWNER_OF( MAKE_GPTR(pt, n) ) == n

TO_LOCAL( MAKE_GPTR(pt, n) ) == pt

TO_GLOBAL(pt)                == MAKE_GPTR( pt, NODE_ID )

IS_LOCAL(gh)                 == SHARE_MEMORY( OWNER_OF(gh), NODE_ID )
```

A.5 Atomic Mailboxes

The atomic `MAILBOX` data type is a language/run-time system supported type that can be used as a mechanism to support critical sections in a dataflow style. An atomic mailbox, among other things, allows multiple producers to send their data, in a non-interfering way, to a single "*merge point*". An atomic mailbox can thus play a role similar to a dataflow non-deterministic merge operator. The operations associated with the `MAILBOX` datatypes are the followings:

`void INIT_MAILBOX( MAILBOX *mb, `*`SLOT_ID`*` item_dropped );`

`void INIT_MAILBOX( MAILBOX *mb, SPTR item_dropped );`
: Initializes the mailbox `mb`. Every time an item is dropped in that mailbox and the item finally reaches the mailbox, then slot `item_dropped` will be signaled.

`void DROP_IN( MAILBOX *GLOBAL mb, void* source, int length );`
: Copies the data (of size `length` bytes) from `source` (a local pointer) into the mailbox. The source data can be safely overwritten as soon as the instruction returns.

`void DROP_IN_SYNC( MAILBOX *GLOBAL mb, void *GLOBAL source,`
`int length, `*`SLOT_ID`*` source_free );`

[30] Except on the MANNA machine, where this is allowed, and in which case the result is of type pointer to `void`.
[31] Again, on MANNA, the result is of type pointer to `void`.

`void DROP_IN_SYNC( MAILBOX *GLOBAL mb, void *GLOBAL source,`
`int length, SPTR source_free );`

Copies the data (of size `length` bytes) from `source` (global handle) into the mailbox `mb`. Signals sync slot `source_free` when the source buffer can be safely overwritten.

`int RETRIEVE_ITEM( MAILBOX mb, void *dest );`

An item is retrieved (and removed) from mailbox `mb` and copied in (the buffer indicated by) `dest`. The operation returns the size (number of bytes) of the retrieved item; it returns 0 if the mailbox was empty. Note that the policy for selecting the item to be retrieved is implementation-dependent.[32]

`int RETRIEVE_ITEM_ADDR( MAILBOX mb, void **dest );`

Similar to `RETRIEVE_ITEM`, except that it copies in `*dest` the address of a buffer containing the retrieved item. Thus, this operation should be used when the size of the item is not known beforehand. Note that is then the responsibility of the client to deallocate this buffer.

`void FREE_MAILBOX( MAILBOX mb );`

Releases the indicated mailbox. The mailbox needs not be empty to be released.

A.6 Obtaining Timing Information

Obtaining correct timing information about a program execution, especially on a parallel machine, generally requires using platform specific operations, e.g., OS-calls, special machine instructions, etc. In order for Threaded-C programs to generate such timing information in a portable and platform-independent way, a new timing data type with related operations has been introduced. The name of this data type is `EARTH_TIME` and it is an abstract (opaque) data type, which means programmers should make no assumption about the representation of this type and should use strictly the constants and operations defined below to manipulate it.

Note that some of the commands described below apply exclusively to the EARTH-MANNA implementation. The items describing these commands, which can be skipped by most readers, have been written in sligthly smaller fonts.

Constants

`EARTH_TIME EARTH_TIME_ZERO`

Used for setting accumulators to 0, e.g., "`EARTH_TIME total_time = EARTH_TIME_ZERO;`".

`double EARTH_TIME_RES`

Resolution of the EARTH clock in seconds. Thus, this is the maximum error of a single reading of the clock (using `EARTH_TIME_READ` below). Programmers should naturally account for error accumulation when adding multiple readings.

`double EARTH_TIME_MAX`

Maximum value of the EARTH clock in seconds. Thus, this is the largest amount of time (in seconds) which can safely be stored in an `EARTH_TIME` variable without overflow.

[32]However, most implementations will probably choose to use a simple FIFO queue.

`EARTH_TIME_LONG`

This is a compile-time constant, not used in regular code. If defined, it tells the compiler to use longer time values. This is for systems in which the time has a small range (e.g., the MANNA, where `EARTH_TIME_MAX<86`).

`double EARTH_TIME_MAX_LONG`

Same as `EARTH_TIME_MAX`, except on systems with short time values in which `EARTH_TIME_LONG` is defined. In such cases, `EARTH_TIME_MAX_LONG` is the maximum number of seconds in the longer time variable type. For systems with sufficiently high ranges, the flag is ignored, and `EARTH_TIME_MAX_LONG` is the same as `EARTH_TIME_MAX`.

Operations

`EARTH_TIME EARTH_TIME_READ( void )`

Reads the current time and returns it in the form of an `EARTH_TIME`. There is no defined "zero time," so all timing should be done in a relative way, that is, by taking the difference between times read at the beginning and end of a timed section.

`EARTH_TIME EARTH_TIME_ADD(EARTH_TIME, EARTH_TIME);`

`EARTH_TIME EARTH_TIME_SUB(EARTH_TIME, EARTH_TIME);`

These add and subtract time values. Presumably, the latter is used for taking the difference between two timestamps, while the former is used for accumulating such differences.

`double EARTH_TIME_NSEC(EARTH_TIME);`

`double EARTH_TIME_USEC(EARTH_TIME);`

`double EARTH_TIME_MSEC(EARTH_TIME);`

`double EARTH_TIME_SEC(EARTH_TIME);`

These convert `EARTH_TIME` values into nanoseconds, microseconds, milliseconds and seconds, respectively. It is assumed these are already differential times (taking the difference between `EARTH_TIME`s at two different times), since converting an absolute `EARTH_TIME` value is meaningless (see above).

`void EARTH_TIME_UPDATE( void );`

This is for systems in which the time has a small range (e.g., the MANNA, where `EARTH_TIME_MAX<86`). It is a hint to the compiler that now would be a good time to check the time and update it internally. It is ignored if `EART_TIME_LONG` is undefined or if the `EARTH_TIME` variable has a sufficient range. It is used in places where the programmer suspects the time between two calls to `EARTH_TIME_READ()` may exceed `EARTH_TIME_MAX`. The system checks the time and internally keeps track of the time since the last call of `EARTH_TIME_READ()`, noting any rollovers. Thus, the next time `EARTH_TIME_READ()` is called, the `EARTH_TIME` (which is in `long` format) is guaranteed correct, even if it exceeds `EARTH_TIME_MAX`, provided it is less than `EARTH_TIME_MAX_LONG`.

(Note: if `EARTH_TIME_LONG` is defined, the implementation may use a `long` time resource if available, in which case both `EARTH_TIME_MAX` and `EARTH_TIME_MAX_LONG` would be large, and `EARTH_TIME_UPDATE()` would simply be ignored. This should only be done if the cost of the additional resource is small. In any case, it is not available in MANNA.)

Index