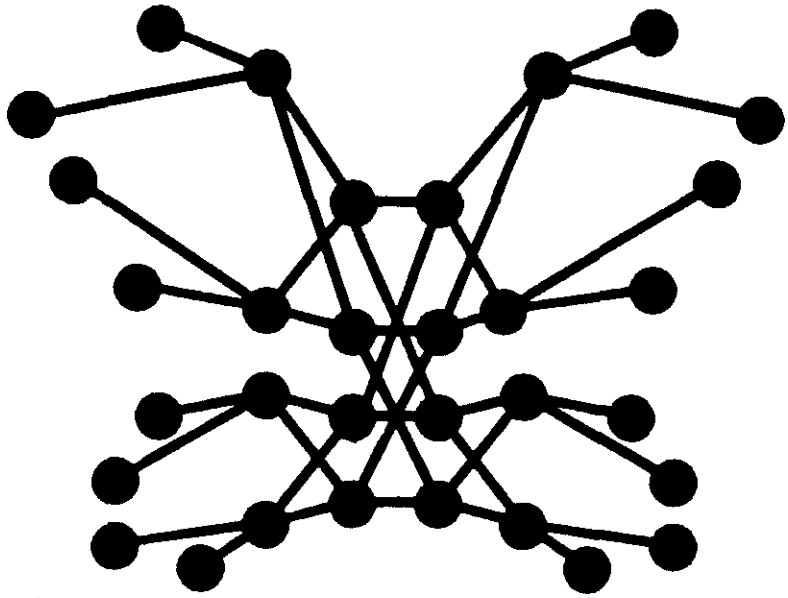


Butterfly Project Report
21

Ant Farm: A Lightweight Process Programming Environment

M.L. Scott and K.R. Jones

August 1988

Computer Science Department
University of Rochester
Rochester, NY 14627

Ant Farm:
A Lightweight Process Programming Environment

Michael L. Scott
Kurt R. Jones

University of Rochester
Department of Computer Science
Rochester, NY 14627

August 1988

ABSTRACT

Many parallel algorithms require a collection of processes whose number is linear (or worse) in the size of the problem to be solved. Few programming environments support such flagrant parallelism. Most often each virtual process of a program corresponds to a single, expensive heavyweight entity supplied by the operating system. Ant Farm is a library package for the BBN Butterfly Parallel Processor that allows programs to split computational effort across many *lightweight* threads of control. On each node of the Butterfly, Ant Farm provides for the execution and management of dozens, even hundreds, of threads. A full set of mechanisms is provided for location-transparent communication, sharing, and synchronization. This report describes the Ant Farm model, its implementation, and its programming interface.

This work was supported in part by NSF CER grant number DCR-8320136 DARPA ETL contract number DACA76-85-C-0001, and an IBM Faculty Development Award

1. Introduction

Ant Farm is a library package developed at the University of Rochester for use on the BBN Butterfly Parallel Processor [2]. It is compatible with both C and Modula-2, and runs at present under BBN's Chrysalis operating system [1]. The goals of Ant Farm are to:

(1) Provide a user-level programming environment that permits the maximum possible number of lightweight processes (threads). The intent is to allow the programmer to choose a level of parallelism based on the needs of the application, rather than on constraints imposed by systems software.

(2) Reduce and simplify the coding overhead necessary to write parallel programs on the Butterfly.

(3) Provide primitives to support a variety of models of coordination and synchronization between threads. These include shared memory, Chrysalis events and dual queues, semaphores, and monitors.

Several factors motivated the development of Ant Farm. A primary conclusion of Rochester's DARPA Benchmark study [4] was that there was a pressing need for a programming system that would support very many more processes than processors. Modula-2, ported to the Butterfly at Rochester [12] provides lightweight threads (coroutines), but their definition admits no true parallelism; each Modula-2 program lives inside a single Chrysalis process. Mechanisms for interaction between Chrysalis processes (e.g. SMP [8, 10]) introduce a new communication model, one that differs radically from the existing mechanism for interaction between coroutines. At the other end of the spectrum, the lightweight task model of the BBN Uniform System [3] provides no mechanism for synchronization between threads other than busy-waiting. Ant Farm was designed to combine the efficiency and high degree of virtual parallelism of the Uniform System with the scheduler-based synchronization of full-fledged processes.

Sections 2 and 3 of this report provide an overview of Ant Farm. Section 4 presents and explains an example application. Section 5 describes our implementation. Appendices 1 and 2 contain copies of the Ant Farm interface description files for Modula-2 and C.

2. The Ant Farm Model

An Ant Farm program consists of a large collection of lightweight threads of control. Initially, at program startup, there is only one thread. Any existing thread can start new threads explicitly at any time. Any existing thread may also choose to terminate itself. The program as a whole terminates when the original thread (or another thread in the same virtual node) runs off the end of its program without explicitly terminating.

Ideally, threads would all run in a single shared address space and would make full use of all available processors at all times. In the context of the Butterfly, however, the code required to produce such complete location transparency would have entailed more complexity and run-time bookkeeping cost than we were willing to accept. We have therefore adopted the simplifying assumption that each thread is confined to a single "virtual node." Virtual nodes share a large, distributed heap in which data structures may be allocated at run time. The heap occupies the

same addresses on every node, so pointers into shared data structures can be passed between nodes without translation. It is usually best to create one virtual node per physical processor, though more or fewer can be used if desired. As described in section 5, each virtual node is implemented by a separate Chrysalis process. Ant Farm will make full use of available processing power only when there is a runnable thread in some virtual node on every physical processor.

A traditional sequential program has several distinct storage classes for memory, including "automatic" variables (allocated on the stack), "own" variables (allocated statically, but visible only locally), global variables (also allocated statically), and dynamic variables (allocated from a heap). In addition, the Ant Farm library defines a new class of *dynamic shared* variables. All of the traditional classes are available in Ant Farm, in the sense that they are recognized by source language compilers. Only automatic and dynamic shared variables, however, are consistent with the Ant Farm model. Regular dynamic variables may also be used, so long as they are allocated and accessed by only a single thread. The use of own and global variables is strongly discouraged.

The role played by the different storage classes should be familiar to any user of the Uniform System. Dynamic shared variables reside in the Ant Farm heap, and are accessible to threads on all virtual nodes. The other, traditional storage classes are accessible only to threads on the *same* virtual node. To the extent that they are of interest to more than one thread, they introduce a notion of semantic locality that Ant Farm is designed to avoid. Automatic variables pose no problem, since they correspond to an individual subroutine activation and thus by implication a thread. Own and global variables can provide the performance-conscious programmer with a cache for global information, but they must be used with care.

To communicate between threads and synchronize access to the shared heap, Ant Farm comes with a variety of supplemental libraries. Those that exist at present are described in the following section. It is expected that others will be created over time. In all cases, synchronization mechanisms are designed to work on a thread-by-thread basis; blocking calls do *not* delay the virtual node as a whole. Ant Farm transfers control to another thread whenever the current thread blocks. To maintain this behavior, it is imperative that all blocking operations be performed through Ant Farm. Programmers should never call blocking Chrysalis operations directly. Spin locks should also be avoided; multiprogramming of physical processors raises the possibility of spinning on a lock held by a suspended virtual node.

An Ant Farm program begins with a call to the EnterAntFarm library routine. EnterAntFarm takes two parameters: the number of virtual nodes in the program and the number of 64 Kbyte segments to be used for the Ant Farm heap. Passing a zero for either of these parameters results in the use of a default: one virtual node and one heap segment for every physical processor in the Chrysalis cluster.

EnterAntFarm must be called directly from the main block of the program (*not* indirectly by a subroutine). Control returns to the original thread after completing initialization. Additional threads can then be created with the StartThread library routine. StartThread allows the user to specify the subroutine in which the new thread should begin execution, the size of its stack in bytes, the virtual node on which it should run (zero indicates the current virtual node), and the

value of a single parameter. The subroutine in which the thread begins should be written without parameters. It must also be declared at the outermost level of lexical nesting. The newly-created thread should call the function ThreadArgs to obtain the fourth parameter to StartThread. ThreadArgs must be called immediately; its return value is unpredictable after the first time the new thread blocks. Under most circumstances it is expected that the initial parameter will be a pointer to an argument block in the shared heap.

Ant Farm also provides a function called WorkSpaceSize that returns the size in bytes of the stack of the current thread. Like ThreadArgs, WorkSpaceSize must be called immediately upon startup, before the thread first blocks. It is useful for programs in which threads create copies of themselves in a branching or pipelined fashion. Only the original thread needs to know the appropriate amount of space to allocate for its children; others simply pass on their own size.

A thread can determine its virtual node number by inspecting the global variable VNodeNum. The total number of virtual nodes can be found in the variable NumVNodes. The initial node is number 1; the others are numbered from 2 to NumVNodes. The standard idiom for starting an Ant Farm thread in Modula-2 is as follows:

```
type argrecptr = pointer to argrec;
var p : argrecptr;  (* arguments pointer *)
. . .
procedure foo;  (* no parameters *)
var args : argrecptr;
begin
   args := argrecptr (ThreadArgs ());
   . . .
   SHdeallocate (args);
   Terminate ();
end foo;
. . .
p := SHallocate (TByteSize (argrec), nodenum);
(* initialize p^ *)
StartThread (foo, WSsize, nodenum, p);
```

In C, the corresponding code looks like this:

```
struct argrec *p;
. . .
foo ()
{
   struct argrec *args = (struct argrec *) ThreadArgs ();
   . . .
   SHdeallocate (args);
   Terminate ();
}
. . .
p = SHallocate (sizeof (argrec), nodenum);
(* initialize *p *)
StartThread (foo, WSsize, nodenum, p);
```

The Terminate routine should usually be called at the end of every thread. A thread that runs off the end of its program without calling Terminate will terminate the entire virtual node. If it is running on the original node, this will terminate Ant Farm as a whole as well (though not very cleanly). A thread than wants to terminate the entire program explicitly can do so by calling the TerminateAntFarm routine.

The SHallocate routine returns the address of a newly-allocated block of contiguous memory in the shared heap. Its two parameters specify the size of the block and its preferred location. SHallocate will choose a location more or less at random if its second parameter is zero or if insufficient space is available on the specified node. The code above attempts to allocate the argument block on the same node as the new thread on the assumption that the thread will be accessing its arguments more often that the creator did. The choice of where to start threads and where to allocate shared memory is entirely in the hands of the programmer; Ant Farm provides no help with the problems of load balancing or locality.

Ant Farm employs a non-preemptive scheduler on each virtual node. Transfers of control occur only when the current thread performs a blocking operation. As a rule, however, it is unwise to count on the lack of preemption to guarantee mutual exclusion. Data in the shared heap can be accessed by threads on multiple physical processors. Even within a single virtual node, it can be difficult to tell when a subroutine call might cause the current thread to block. Several of the basic Ant Farm operations contain block points, even though they are not explicitly related to synchronization.

One consequence of the lack of preemption is that a thread performing low-priority "background" computation can starve the other threads on its node. Ant Farm provides a routine called Yield that allows the current thread to relinquish the processor temporarily in favor of some other runnable thread. Threads that have been blocked for a synchronization event that has since occurred are considered "runnable." Placing a call to Yield in the outer loop of a background thread allows it to poll for the runnability of other threads on its virtual node.

Chrysalis-style exception handling (with catch and throw) is available in C, though not in Modula-2. Ant Farm routines are written not to throw (internal catch blocks arrange to return error statuses instead). A C program that uses throws should include catch blocks around the bodies of functions in which threads begin execution. A throw that escapes the original scope of a thread is likely to crash the virtual node.

3. Synchronization Mechanisms

Ant Farm currently provides four different synchronization libraries, for semaphores, events, dual queues, and monitors. Dual queues can be used in conjunction with the Ant Farm heap to provide several forms of message passing. Additional libraries could easily be written to provide such additional mechanisms as path expressions, serializers, and conditional critical regions. The four existing libraries are described in the subsections below. Complete specifications of their C and Modula-2 interfaces are contained in the Appendix. The facilities for implementing additional libraries are described in section 5.

3.1. Semaphores

Semaphores are one of the oldest synchronization mechanisms, and were first described by Dijkstra [5]. A semaphore has an integer value, and a pair of operations called P and V that can be used to change its value. (The letters P and V are mnemonic in Dutch, but that doesn't help much.) Both P and V take a single argument, the name of the semaphore. They operate atomically. Their actions are defined as follows:

P: decrement the value of the semaphore;
if the result is negative, then wait.

V: increment the value of the semaphore;
if the result is negative or zero, then unblock a waiting thread.

A semaphore whose value is always less than or equal to one (a so-called *binary* semaphore) can be used as a simple lock. Threads perform P operations to acquire the lock, V operations to release the lock.

In addition to P and V, Ant Farm provides three additional semaphore operations: MakeSem, DestroySem, and SemValue. MakeSem takes two arguments. The first is the maximum number of threads that can ever wait on the semaphore at once. The second is the semaphore's initial value. MakeSem returns a zero if no more semaphores can be created. SemValue returns the current value of the semaphore. DestroySem fails fatally if any thread is waiting on the semaphore. P and V fail fatally if the limit given to MakeSem is ever exceeded.

3.2. Events

Events are the basic synchronization primitive in Chrysalis. An event is similar to a binary semaphore. It provides two operations, WaitEvent and PostEvent, that resemble P and V, respectively. The differences are: (1) though any process can post an event, only the owner of the event can wait, and (2) a 32-bit datum can be provided to the post operation, to be returned by a subsequent wait. The effect of the first difference on Ant Farm is that an event is owned by a particular virtual node, and only threads in that node can wait on it.

In addition to WaitEvent and PostEvent, Ant Farm provides three additional event operations: MakeEvent, DestroyEvent, and PollEvent. MakeEvent takes one argument: the virtual node that will own the event. (If this argument is zero then the event is created on the current virtual node.) It returns a zero if no more events can be created. DestroyEvent fails fatally if some thread is waiting for the event. PollEvent is a non-blocking version of WaitEvent. It returns a Boolean value. It takes as arguments both the name of the event and the name of a variable into which a result may be written. If the event has been posted, it returns true and fills in the result with the value that would have been returned by WaitEvent. Otherwise it returns false.

3.3. Dual Queues

Dual queues are a Chrysalis-supported generalization of events. The name is derived from the implementation, in which buffers serve a dual purpose, holding either data or events. From the Ant Farm user's point of view, a dual queue is a queue of bounded length with two basic operations: EnqueueDualQueue and DequeueDualQueue. Each entry in the queue can hold one

32-bit datum. The enqueue operation fails fatally if the queue is full. The dequeue operation blocks if the queue is empty, and fails fatally if the number of waiting threads exceeds the size of the queue.

Four additional operations are provided: MakeDualQueue, DestroyDualQueue, PollDualQueue, and DualQueueCount. MakeDualQueue takes one parameter: the maximum number of items to be held in the queue, which is also the maximum number of threads that can wait when the queue is empty. It returns a zero if no more queues can be created. DestroyDualQueue fails fatally if any thread is waiting to dequeue data. PollDualQueue is a non-blocking version of DequeueDualQueue. It returns a Boolean value. It takes two arguments: the name of the dual queue and the name of a variable into which a result may be written. If the queue is non-empty, it returns true and fills in the result with the value that would have been returned by DequeueDualQueue. Otherwise it returns false. The DualQueueCount operation returns the number of items in the queue or, if the queue is empty, the negative of the number of threads waiting to dequeue from it. Beware that a positive result does not necessarily mean that an immediately-subsequent dequeue operation will succeed without blocking; a thread in some other virtual node may intervene.

3.4. Monitors

Monitors in Ant Farm provide the classic Hoare-style semantics [6] in a style reminiscent of monitored records in Mesa [11]. Conceptually, a monitor is like a locked room containing shared data. Only one thread is allowed inside the room at a time. The two most basic operations are therefore EnterMonitor and LeaveMonitor. EnterMonitor will block if the monitor is busy.

The advantage of monitors over simple locks is that they make it possible for threads to synchronize on conditions that are more sophisticated than simple mutual exclusion. The creator of a monitor may specify that it is to contain a number of **condition variables** for synchronization. Condition variables support two basic operations: WaitCondition and SignalCondition. WaitCondition always blocks. SignalCondition unblocks a waiting thread if there is one, and does nothing otherwise. The principal difference between condition variables and semaphores is that the V-like signal operation is lost if no thread is waiting.

In keeping with Hoare's definition, a thread that is unblocked by a signal operation reacquires the monitor immediately. The thread that performed the signal operation steps outside temporarily and blocks. If there are threads awaiting entry to the monitor when the thread that is inside leaves (via LeaveMonitor or WaitCondition), the longest-waiting signaller is unblocked. If there are no waiting signallers, the thread that has been waiting longest for the EnterMonitor operation is unblocked instead.

In addition to EnterMonitor, LeaveMonitor, WaitCondition, and SignalConsition, Ant Farm provides three additional operations: MakeMonitor, DestroyMonitor, and ConditionCount. MakeMonitor takes two parameters: the number of condition variables to be associated with the monitor and the maximum number of threads that will ever use the monitor at once. It returns a zero if no more monitors can be created. DestroyMonitor fails fatally if any thread is waiting to enter the monitor or waiting on one of its condition variables. ConditionCount returns the

number of threads currently waiting on the condition. All three condition variable operations (SignalCondition, WaitCondition, and ConditionCount) take two parameters: the name of the monitor and the sequence number of the condition. WaitCondition, SignalCondition, and EnterMonitor fail fatally if the limit given to MakeMonitor is ever exceeded. The condition variables of a given monitor are numbered starting at zero.

Ideally one would support monitors with a language that prevented access to the data of a monitor by threads that are outside. Unfortunately, there appears to be no way to achieve this goal in the context of a library package. It is thus the responsibility of the Ant Farm programmer to ensure that monitors are used correctly. The standard idiom for monitored data is to place it in a record whose first field is the monitor itself:

```
type fooptr = pointer to foo;
type foo = record (* monitored *)
    lock : Monitor;
    data : . . .
end;
```

Any operation that accesses the data should be bracketed with EnterMonitor and LeaveMonitor operations:

```
var x : fooptr;
. . .
x := fooptr (SHallocate (Tsize (foo), nodenum));
x^.lock := MakeMonitor (numconds, numthreads);
. . .
EnterMonitor (x^.lock);
with x^ do
    (* access data *)
end;
LeaveMonitor (x^.lock);
. . .
DestroyMonitor (x^.lock);
```

While accessing the data, signal and wait operations are straightforward:

```
SignalCondition (x^.lock, condnum);
. . .
WaitCondition (x^.lock, condnum);
```

The corresponding C code looks very much the same:

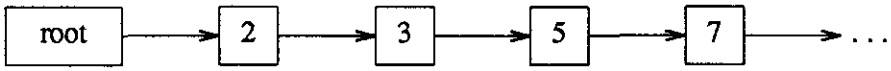
```
struct foo {
    Monitor lock;
    . . .
} *x;

x = (struct foo *) SHallocate (sizeof (struct foo), nodenum));
x->lock = MakeMonitor (numconds, numthreads);

EnterMonitor (x->lock);
. . .
```

For the sake of readability, conditions should be identified by named constants.

4. A Simple Example

Figures 1 and 2 contain an Ant Farm implementation of the Sieve of Eratosthenes, the classic algorithm to compute prime numbers [7, p. 394]. The sieve is not a particularly good algorithm for generating primes from a computational point of view. It is also not a very good example of parallel programming (at least not in the form presented here), because the amount of computation performed between communication events is too small to make effective use of even the lightest of lightweight threads. On the other hand, the algorithm is non-trivial in the sense that it would take many, many lines of code to implement directly on top of Chrysalis. It is also not an obvious candidate for the Uniform System, since its threads require a great deal of synchronization. It displays good speedup as processors are added. It makes a good tutorial.

The basic idea behind the algorithm is displayed in the following picture:



The threads of the algorithm constitute a pipeline, each element of which contains a single prime. The root thread feeds all of the natural numbers (except 1!) into the pipeline one at a time. Each subsequent thread keeps the first number it receives (a prime), and passes through only those later numbers that are not divisible by the first.

The code begins at line 66 (line 59 in C) with a call to EnterAntFarm. The two zero parameters arrange for the default of one virtual node and heap segment per physical processor. The next two lines create a thread to print the primes as they are discovered, together with a dual queue to hold the inputs to that thread. The zero parameters in the call to StartThread indicate that the printer thread has no startup parameters and that it is to run on the same virtual node as the root thread. The printQ variable is shared between the root and printer threads, illustrating a legitimate use of static global variables. The root thread ignores the return values from its calls to resource-allocating subroutines. This is probably not good programming practice, but is almost certainly safe, since no other demands on those resources have yet been made.

The code between lines 70 and 75 (63 and 68 in C) creates the first thread in the pipeline (the one that will hold the number 2). The argument block for that thread contains the name of the dual queue for the printer thread. It also contains the names of two events: one of which is used to pass numbers into the pipeline and the other of which is used for handshaking, to acknowledge the receipt of a number and permit the sender to continue. (Without such flow control, it would be possible for a thread to crash the program by posting an event a second time before its owner had awaited it.) The last three lines of the main program keep feeding numbers into the pipeline until it reaches MAXINT (which it won't) or until code somewhere else causes termination (which it will).

The real work of the program occurs in the Eratosthenes procedure, which is executed by the threads of the pipeline. Each thread begins by acquiring its arguments from the built-in ThreadArgs function. (Note that the Eratosthenes routine itself has no arguments.) In lines 30 and 31 (21 and 22 in C) the thread acquires its prime, unblocks its predecessor (which has been

```
module sieve;

from system import TByteSize, MAXINT;
from io import writef, output;
from AntFarm import EnterAntFarm, StartThread, NumVNodes, VNodeNum,
    ThreadArgs, TerminateAntFarm;
from Events import Event, MakeEvent, PostEvent, WaitEvent;
from DualQueues import Queue, MakeDualQueue, EnqueueDualQueue, DequeueDualQueue;
from SHeap import SHallocate;

const WSsize = 2000; (* bytes *)
type argblkptr = pointer @nocheck to argblock;
    argblock = record
        forward, back : Event;
        home : Queue;
    end;

procedure NextNode (n : cardinal) : cardinal;
begin
    return (n mod NumVNodes) +1;
end NextNode;

procedure Eratosthenes;
var
    args, succ : argblkptr;
    myprime, n, junk, nextVnode : cardinal;
begin
    succ := nil;
    args := argblkptr (ThreadArgs ());
    myprime := WaitEvent (args^.forward); PostEvent (args^.back, 0);
    EnqueueDualQueue (args^.home, myprime);
    loop
        n := WaitEvent (args^.forward); PostEvent (args^.back, 0);
        if n mod myprime <> 0 then
            if succ = nil then
                nextVnode := NextNode (VNodeNum);
                succ := SHallocate (TByteSize (argblock), nextVnode);
                succ^.forward := MakeEvent (nextVnode);
                succ^.back := MakeEvent (0);
                succ^.home := args^.home;
                if not StartThread (Eratosthenes, WSsize, nextVnode, succ) then
                    TerminateAntFarm ();
                end;
            end;
            PostEvent (succ^.forward, n); junk := WaitEvent (succ^.back);
        end;
    end;
end Eratosthenes;

var printQ : Queue;
```

```
51
52 procedure printer;
53 var n : cardinal;
54 begin
55     loop
56         n := DequeueDualQueue (printQ);
57         writef (output, "%d\n", n);
58     end;
59 end printer;
60
61 var head : argblkptr;
62     nextVnode, i, junk : cardinal;
63     void : Boolean;
64
65 begin (* main *)
66     EnterAntFarm (0, 0);
67     printQ := MakeDualQueue (NumVNodes);
68     void := StartThread (printer, WSsize, 0, 0);
69
70     nextVnode := NextNode (VNodeNum);
71     head := SHallocate (TByteSize (argblock), nextVnode);
72     head^.forward := MakeEvent (nextVnode);
73     head^.back := MakeEvent (0);
74     head^.home := printQ;
75     void := StartThread (Eratosthenes, WSsize, nextVnode, head);
76
77     for i := 2 to MAXINT do
78         PostEvent (head^.forward, i); junk := WaitEvent (head^.back);
79     end;
80 end sieve.
```

Figure 1: Modula-2 Code for the Sieve of Eratosthenes

waiting for the acknowledgment), and enters the prime on the printer's dual queue. In the subsequent loop the thread repeatedly receives a number from its predecessor, checks it for divisibility by its prime, and passes it on to its successor if appropriate. The if statement at line 35 (line 26 in C) creates the successor on demand the first time it is needed. Successive elements of the pipeline are placed on successive virtual nodes, with wrap-around. The program terminates when insufficient resources exist to create another thread.

5. Implementation Details

5.1. Memory Configuration

Each Ant Farm virtual node is implemented by a (heavyweight) Chrysalis process. Each virtual node has the potential of running many threads, limited principally by available local memory. Since each virtual node consumes Chrysalis resources, the most efficient use of

```
 1 #include "AntFarm.h"
 2 #include "Events.h"
 3 #include "DualQueues.h"
 4 #include "SHeap.h"
 5
 6 #define WSsize 2000  /* bytes */
 7
 8 struct argblk {
 9     Event forward, back;
10     Queue home;
11 }
12
13 #define NextNode(n) (n % NumVNodes + 1)
14
15 Eratosthenes ()
16 {
17     struct argblk *args = ThreadArgs ();
18     struct argblk *succ = 0;
19     unsigned myprime, n, junk, nextVnode;
20
21     myprime = WaitEvent (args->forward); PostEvent (args->back, 0);
22     EnqueueDualQueue (args->home, myprime);
23     for (;;) {
24         n = WaitEvent (args->forward); PostEvent (args->back, 0);
25         if (n % myprime) {
26             if (!succ) {
27                 nextVnode = NextNode (VNodeNum);
28                 succ = (struct argblk *)
29                     SHallocate (sizeof (struct argblk), nextVnode);
30                 succ->forward = MakeEvent (nextVnode);
31                 succ->back = MakeEvent (0);
32                 succ->home = args->home;
33                 if (!StartThread (Eratosthenes, WSsize, nextVnode, succ)) {
34                     TerminateAntFarm ();
35                 }
36             }
37             PostEvent (succ->forward, n); junk = WaitEvent (succ->back);
38         }
39     }
40 }
41
42 Queue printQ;
43
44 printer ()
45 {
46     unsigned n;
47
48     for (;;) {
49         n = DequeueDualQueue (printQ);
50         printf ("%d\n", n);
```

```
51      }
52  }
53
54  struct argblk *head;
55  unsigned nextVnode, i, junk;
56
57  main ()
58  {
59      EnterAntFarm (0, 0);
60      printQ = MakeDualQueue (NumVNodes);
61      (void) StartThread (printer, WSsize, 0, 0);
62
63      nextVnode = NextNode (VNodeNum);
64      head = SHallocate (sizeof (struct argblk), nextVnode);
65      head->forward = MakeEvent (nextVnode);
66      head->back = MakeEvent (0);
67      head->home = printQ;
68      (void) StartThread (Eratosthenes, WSsize, nextVnode, head);
69
70      for (i = 2; i != 0xffffffff; i++) {
71          PostEvent (head->forward, i); junk = WaitEvent (head->back);
72      }
73  }
```

Figure 2: C Code for the Sieve of Eratosthenes

memory, cycles, and virtual address space is obtained when exactly one virtual node runs on every physical processor in the user's cluster.

All virtual nodes have the same number of segments at the same addresses. Since instruction fetching through the switch is unthinkable, virtual nodes on separate processors use separate copies of the code. For compatibility with Chrysalis, virtual nodes on the same physical processor use separate copies as well. The data segments of separate virtual nodes are also distinct, because they contain large amounts of node-specific data in the run-time library for the management of threads and events. All shared data is in the Ant Farm heap, created by EnterAntFarm and mapped into the same virtual addresses on each virtual node.

EnterAntFarm takes two arguments, the desired number of virtual nodes and the desired number of 64 Kbyte heap segments. The number of segments in the heap is rounded up to the next even multiple of the number of virtual nodes. For both arguments a zero means one per physical processor. In the original (root) node, EnterAntFarm (1) creates, maps in, and initializes the shared heap; (2) builds a number of descriptive data structures in the heap; (3) starts an appropriate number of additional nodes (clones), passing them the addresses of the data structures; (4) creates a pair of background threads (discussed below); (5) waits for all clones to complete initialization; and (6) returns.

Since the clones are exact copies of the root process, they too begin execution by calling EnterAntFarm. Once inside, however, they perform a different set of operations. Each clone (1) reads its command-line arguments to find its virtual node number and the address and object ids of the initial segment of the shared heap; (2) maps in the initial heap segment and reads out of it the rest of its startup parameters, including the addresses and object ids of the rest of the segments of the shared heap; (3) maps in the remainder of the heap; (4) participates if necessary in the creation of additional virtual nodes; (5) creates its background threads; (6) informs the root node that it has finished initialization; and (7) calls TerminateThread to end execution of its main thread. This last step causes the clone to fall into its thread dispatcher, waiting for external events.

The creation of clone processes is parallelized to minimize the time required for Ant Farm initialization. The root process creates a dual queue on which it enqueues, for each other virtual node to be created, the processor number on which that node should run. Each process upon creation (the root included) repeatedly dequeues a processor number and creates a virtual node, stopping only when the queue is empty. This process-creation tree is similar to that provided by Rochester's Crowd Control package [9] except that a given process may start an arbitrary number of children, instead of only two.

5.2. Thread Management

Within a single virtual node, each Ant Farm thread is implemented as a coroutine, with stack space allocated out of the local (malloc) heap. All transfers between coroutines occur inside the Ant Farm library, and are invisible to the user. Users should generally assume that threads all run in parallel. There are two noteworthy exceptions: (1) a thread that never blocks will starve its peers, and (2) if used with care, the lack of pre-emption can provide mutual exclusion for access to static data of the virtual node. The ThreadArgs and WorkSpaceSize routines rely on this mutual exclusion, and must be called by a newly-created thread before the first time that it blocks.

The following Ant Farm calls have the potential to block: StartThread, TerminateThread, Yield, MakeEvent, WaitEvent, P, DequeueDualQueue, EnterMonitor, WaitCondition, and SignalCondition. All of the blocking primitives call WaitEvent internally. When the current thread attempts to wait on an event that has not yet been posted, a context switch must take place to enable another runnable thread within the virtual node. The registers of the currently running thread are saved on its stack, a new runnable thread is selected, the registers of the new thread are loaded from its stack, and execution resumes in the new thread.

To facilitate the scheduling of threads, Ant Farm maintains a ready list of runnable threads and a mapping between blocked threads and the events on which they are waiting. When the current thread must block it adds an event/thread pair to the event dictionary and invokes the Ant Farm scheduler. When a StartThread call is made, the currently running thread is placed on the ready list (since it is still runnable) and a context switch is made to the new thread. Control will return only when the new thread blocks.

There does not always exist a runnable thread when a context switch has to be made — the ready list may be empty. In this case the virtual node must actually block to Chrysalis, waiting

for any of its events to be posted. Upon awakening it locates the posted event in the event dictionary and resumes the corresponding thread.[1] If there is no corresponding thread (i.e. the event has been posted before it was required), then an entry is made in the event dictionary (with a null thread id) indicating that the event is already posted. When the target thread would normally block to wait for the event no blocking actually occurs, as the event is immediately available.

5.3. Background Threads

In every virtual node, including the root, EnterAntFarm creates two background threads that continue to exist throughout the life of the program. The purpose of these threads is to receive and handle requests from other virtual nodes to create local threads and events. Since EnterAntFarm returns to the main thread only on the root node, these threads provide the only means for future execution on nodes other than the root. Each background thread begins execution by creating a dual queue on which it waits for requests to be enqueued. Both StartThread and MakeEvent check to see whether they are to perform their work locally or remotely. In the latter case they enqueue a request on the appropriate dual queue and wait for the request to be honored. Since the remote node may not respond for an arbitrary amount of time (its threads may never block), the waits in StartThread and MakeEvent may cause a transfer of control to another runnable thread.

The requests enqueued by MakeEvent simply contain the name of the event on which the requesting thread is waiting. The requests enqueued by StartThread are actually pointers to argument blocks in the shared heap. Each argument block contains the address of the subroutine in which the new thread is to begin execution, the stack size for that thread, the name of the event on which the requesting thread is waiting, and a pointer to the parameter block supplied by the user (to be provided to the new thread via ThreadArgs).

To avoid the overhead of allocation from the shared heap on every StartThread call, Ant Farm caches argument blocks for reuse. It also caches events instead of invoking the Chrysalis operations to destroy and re-create them. Events are used not only by user programs, but also internally for control and handshaking during operations requiring synchronization. This optimization therefore increases the performance of many operations.

5.4. Caveats

(1) As previously mentioned, a thread may wait only on events owned by the virtual process in which it runs. Fatal errors will result if a thread attempts to use an event belonging to another virtual node.

(2) Posting to an event that is already posted will cause a fatal error. Thus, Ant Farm programs communicating with events must usually use handshaking strategies (e.g. a pair of events) to ensure that an event is not posted again until the target thread has seen the

[1] This dispatching mechanism is borrowed from the Lynx distributed programming language [13, 14], where it was used in the absence of shared memory.

original posting. This limitation does not occur with the other synchronization mechanisms (e.g. dual queues, semaphores, and condition variables), since they incorporate storage space for multiple postings.

(3) It is important to end the code of every thread with an explicit call to TerminateThread. Otherwise the virtual node as a whole will terminate when the thread runs off the end.

(4) The stack size parameter specified in any StartThread call must be big enough to hold the entire stack the thread will ever use. This can be difficult to predict, especially in the presense of recursive subroutines. Use of a stack that is not big enough may cause the Butterfly node to crash or hang when the stack overflows. The minimal stack size required for even the most basic of threads appears to be approximately 2000 bytes; use more stack when in doubt.

(5) The scarcity of segment attribute registers (SARs) on the Butterfly imposes a restriction of at most 256 objects mapped into the address space of a heavyweight process. Shared memory is thus limited in size. With a 100-node cluster, each virtual node can hold at most two segments of the shared heap. Since our implementation does not allow heap objects to span nodes, it will not be possible to allocate anything larger than 128 Kbytes.

(6) There is a limit on the number of events that can be created by an Ant Farm program. Like the storage available from malloc, this limit varies with a large number of factors, including the size of the cluster, the number of virtual nodes, and the number of other Chrysalis objects allocated on the machine. Our experience suggests that for typical programs the limit on stack space for threads is likely to be encountered before the limit on events.

(7) Additional synchronization mechanisms can be added easily to Ant Farm. Internally, however, they must all be implemented in terms of other, existing mechanisms (primarily events). A call to a blocking operation in Chrysalis (e.g. Sleep) will block the entire virtual node, not just the calling thread.

Appendix 1: Modula-2 Interface Definitions

1.1. AntFarm.def

```
definition module AntFarm;

(*****************************************************************
  Major AntFarm definitions.
*****************************************************************)

from SYSTEM import ADDRESS;
export  (* var *) VNodeNum, NumVNodes,
        (* proc *) EnterAntFarm, TerminateAntFarm, StartThread,
                   TerminateThread, Yield, ThreadArgs, WorkSpaceSize;

VAR NumVNodes : CARDINAL;
```

```
(* Set by EnterAntFarm to the number of virtual nodes running. *)

VAR VNodeNum : CARDINAL;
(* Set by EnterAntFarm to the virtual node number of the caller (a number
   between 1 and NumVNodes). *)

procedure EnterAntFarm(VirtualNodes : CARDINAL; VirtualHeapWidth : CARDINAL );
(* Should be the first AntFarm library call in any AntFarm pgm.  Starts
   "VirtualNodes" heavyweight processes across the nodes of the cluster (use
   VirtualNodes=0 to establish 1-1 mapping btw real and virtual nodes).
   Also sets up "VirtualNodes" shared heaps (one per Virtual node), each of
   size (VirtualHeapWidth/VirtualNodes) x 64k bytes.  (Use VirtualHeapWidth=0
   to establish one 64k heap per virtual node.)
     Returns to caller if root.  If caller is a child does not return. *)

procedure TerminateAntFarm();
(* Stop AntFarm *)

procedure StartThread(name : proc; WSsize : CARDINAL;
                  VirtualNodeNum : CARDINAL; HeapArgBlock : ADDRESS) : BOOLEAN;
(* Start a thread (a parameterless procedure named "name") on virtual node
   "VirtualNodeNum".  A stack of size "WSsize" bytes is created for the new
   thread (caveat - use WSsize >= 2000). HeapArgBlock is starting address of
   a shared memory block allocated by the user and initialized with whatever
   parameters are needed by the new thread (see ThreadArgs() below).
     Note that "VirtualNodeNum"=0 causes the new thread to be started on the
   caller's virtual node.
     Returns T if thread started ok, else F (e.g. no memory). *)

procedure TerminateThread( );
(* Causes the calling thread to be marked for deletion and a context switch to
   any other runnable thread.  Once the context switch is made, the terminated
   thread is killed. *)

procedure Yield( );    (* temporarily make calling thread LOW priority *)
(* Check if there are any other threads or jobs waiting to run on this
   virtual node.  If so, transparently relinquish control. If not, return to
   caller immediately. *)

procedure ThreadArgs() : ADDRESS;
(* Retrieve the address of the argument block to a new thread. *)

procedure WorkSpaceSize() : CARDINAL;
(* Return the workspace/stack size of the calling thread. *)

end AntFarm.
```

1.2. SHeap.def

```
definition module SHeap;
```

```
(*************************************************************
  Shared Heap operations.  (See 'EnterAntFarm' in AntFarm.def for
  information about the initialization of shared heaps.)  This
  module provides routines for allocation and deallocation of
  blocks of shared memory from shared heaps.
*************************************************************)

from System import ADDRESS;
export (* proc *) SHallocate, SHdeallocate, InitHeap;

procedure SHallocate(size : CARDINAL; vnode : CARDINAL) : ADDRESS;
(* Allocate a contiguous block of shared memory "size" bytes long from the
   heap on virtual node "vnode".  If unavailable, try to allocate from ANY
   other heap.  Use of vnode=0 performs similarly but requires no prefered
   vnode (an arbitrary vnode is selected).
   Address of the beginning of the shared memory block is returned. *)

procedure SHdeallocate(a : ADDRESS);
(* Frees a shared memory block at address "a". This must have
   been previously obtained via SHallocate. *)

procedure InitHeap(vnode : CARDINAL; numsegs : CARDINAL);
(* For internal AntFarm use only. *)

end SHeap.
```

1.3. Semaphores.def

```
definition module Semaphores;

(*************************************************************************
  Semaphore operations.
*************************************************************************)

export (* type *) Sem,
       (* proc *) MakeSem, DestroySem, P, V, SemValue;

type Sem;
(* Opaque type designating a semaphore. *)

procedure MakeSem( NumUserThreads : CARDINAL;
                   initval  : CARDINAL) : Sem;
(* Create and return a new semaphore [with initial value "initval"] to be used
   by (at most) "NumUserThreads" threads. *)

procedure DestroySem(semid : Sem);
(* Delete a semaphore.  *)

procedure V(semid : Sem);
(* Release i.e. increment semaphore value by 1. *)
```

```
procedure P(semid : Sem);
(* Acquire i.e. wait (relinquish control) until semaphore value is
   greater than zero, then decrement semaphore value by 1 and return. *)

procedure SemValue(semid : Sem) : INTEGER;
(* Return current semaphore value. *)

end Semaphores.
```

1.4. Events.def

```
definition module Events;

(*************************************************************************
  Event operations.
*************************************************************************)

export  (* type *) Event,
        (* proc *) MakeEvent, DestroyEvent, PostEvent, WaitEvent, PollEvent;

type Event; (* opaque type designating an Event *)

procedure MakeEvent(VirtualNodeNum : CARDINAL) : Event;
(* Create and return an event on Virtual node "VirtualNodeNum".
   Zero indicates current virtual node. *)

procedure DestroyEvent(AnEvent : Event);
(* Delete existing event. *)

procedure PostEvent(AnEvent : Event; Data : CARDINAL);
(* Post data to a specific event. *)

procedure WaitEvent(MyEvent : Event) : CARDINAL;
(* Await the posting of an event, transparently relenquishing control.
   Return event data. *)

procedure PollEvent(AnEvent : Event; VAR data : CARDINAL) : BOOLEAN;
(* Same as WaitEvent except returns false if event not immediately available
   i.e. NONBLOCKING and won't relinquish control to another thread...
   Return boolean T if event was posted, with VAR data set. *)

end Events.
```

1.5. DualQueues.def

```
definition module DualQueues;

(*************************************************************************
  Dual Queue operations.
*************************************************************************)

export (* type *) Queue,
```

```
       (* proc *) MakeDualQueue, DestroyDualQueue, EnqueueDualQueue,
                   DequeueDualQueue, PollDualQueue, DualQueueCount;

type Queue;
(* Opaque type designating a Dual Queue. *)

procedure MakeDualQueue(numitems : CARDINAL) : Queue;
(* Create and return a new Dual queue that can hold (at most) "numitems"
   elements. *)

procedure DestroyDualQueue(qid : Queue);
(* Delete a Dual Queue.  *)

procedure EnqueueDualQueue(qid : Queue; data : CARDINAL);
(* Enqueue a datum on a Dual Queue.  *)

procedure DequeueDualQueue(qid : Queue) : CARDINAL;
(* Dequeue a datum from a Dual Queue.  If Dual Queue is empty, then
   wait (relinquish control) and regain control once data becomes available.
   Data is set on return. *)

procedure PollDualQueue(qid : Queue; VAR data : CARDINAL) : BOOLEAN;
(* Attempt to dequeue a datum from Dual Queue, but do not relinquish control
   if the Dual Queue is empty. Return false if no data available, otherwise
   return true with data variable set. *)

procedure DualQueueCount(qid : Queue) : INTEGER;
(* If the Dual Queue contains data, return the number of items in it. If one
   or more threads are waiting for data to become available (i.e. the queue
   contains only events), return a negative value indicating the number of
   waiting threads. If the Dual Queue is empty, return zero. *)

end DualQueues.
```

1.6. Monitors.def

```
definition module Monitors;

(*************************************************************************
  Monitor operations.
*************************************************************************)

export (* type *) Monitor,
       (* proc *) MakeMonitor, DestroyMonitor, EnterMonitor, LeaveMonitor,
                  WaitCondition, SignalCondition, ConditionCount;

type Monitor;
(* Opaque type designating a monitor. *)

procedure MakeMonitor( NumConditions : CARDINAL;
                       NumUserThreads : CARDINAL) : Monitor;
```

```
(* Create and return a new monitor.  "NumConditions" specifies the number of
   condition queues in the monitor; "NumUserThreads" specifies the maximum
   number of threads that can simultaneously use the monitor.
     Note that WaitCondition, SignalCondition, and ConditionCount use
   condition values numbered from 0 to NumConditions-1. *)

procedure DestroyMonitor(monid : Monitor);
(* Delete a monitor. *)

procedure EnterMonitor(monid : Monitor);
(* Enter a monitor. If the monitor is busy or there are other threads
   on the entry queue ahead of it, wait (relinquish control) until it
   is caller's "turn".  When control is returned to the caller, it is
   guaranteed to be the only thread executing within the monitor. *)

procedure LeaveMonitor(monid : Monitor);
(* Exit a monitor, releasing exclusive control over it. *)

procedure WaitCondition(monid : Monitor; condit : CARDINAL);
(* Release exclusive control over the monitor and wait (relinquish control)
   for the specified condition on the appropriate condition queue.  "Condit"
   is a value between 0 and NumConditions-1.
   Control is returned when the calling thread becomes first on the
   specified condition queue and the condition becomes true (i.e. some
   thread signals the condition). *)

procedure SignalCondition(monid : Monitor; condit : CARDINAL);
(* If no thread is waiting on the specified condition queue, simply return.
   Otherwise, release exclusive control over the monitor, wait (relinquish
   control) on the urgent queue, and wake up the appropriate waiting thread.
   Control is returned when the calling thread becomes first on the urgent
   queue and some thread executes a LeaveMonitor or WaitCondition call. *)

procedure ConditionCount(monid : Monitor; condit : CARDINAL) : CARDINAL;
(* Return the number of waiting threads in the specified condition queue. *)

end Monitors.
```

Appendix 2: C Interface Definitions

2.1. AntFarm.h

```
extern unsigned NumVNodes, VNodeNum;
/*  The virtual nodes of an Ant Farm program are numbered from 1 to NumVNodes.
    VNodeNum contains a different number on each virtual node, and can be
    inspected by a thread to find out where it is. */

extern void EnterAntFarm ();
/*  unsigned VirtualNodes, VirtualHeapWidth;
        Initializes Ant Farm.  Must be called at the beginning of
```

```
    the program.  Arranges for the specified number of virtual nodes
    (including the original, which will be node number 1).  Creates
    the shared heap, which will consist of VirtualHeapWidth segments
    (rounded up to the next multiple of VirtualNodes) of size 64 Kbytes
    each.  Virtual nodes will be scattered as evenly as possible across
    the physical processors of the user's partition.  If either VirtualNodes
    or VirtualHeapWidth is zero, EnterAntFarm will use a default value
    of one virtual node or heap segment, respectively, per physical
    processor.  Because of the initialization technique employed to
    allocate virtual nodes, command-line invocation parameters for
    Ant Farm will be available (via argc, argv, and envp) on the original
    node only. */

extern void TerminateAntFarm ();

extern char /* boolean */ StartThread ();
/*  void name ();
    unsigned WSsize, VirtualNodeNum;
    char *HeapArgBlock;
        Creates a new thread running on virtual node VirtualNodeNum and
    executing function 'name'.  Returns false if insufficient resources
    exist to create the thread, true otherwise.  The thread will be
    provided with WSsize bytes of stack space.  The thread can obtain
    the final HeapArgBlock parameter by calling ThreadArgs ().  Programmers
    will generally want this parameter to be a pointer into the shared
    heap, though any 4-byte quantity can be used. */

extern void TerminateThread ();

extern void Yield ();
/*  Ant Farm uses a non-preemptive scheduler.  Yield causes the current
    thread to give up the processor temporarily in favor of some other
    runnable thread.  Threads that were blocked for a synchronization event
    that has since occurred are considered "runnable." */

extern char* ThreadArgs ();
/*  Returns the HeapArgBlock parameter from the StartThread invocation that
    caused the creation of the current thread.  Must be called before that
    thread first blocks.  Since it is difficult to tell when a function call
    might block (MakeEvent does, for example, for non-obvious reasons),
    threads should call ThreadArgs immediately, before performing any
    other computation. */

extern unsigned WorkSpaceSize ();
/*  Returns the stack size of the current thread, in bytes.  Returns
    the TOTAL size, not the size remaining.  Like ThreadArgs, must be
    called before the thread first blocks.  Useful for creating copies
    of the current thread, without keeping track of sizes explicitly. */
```

2.2. SHeap.h

```
extern char* SHallocate ();
/*  unsigned size, vnode;
        Allocates a contiguous block of shared memory of the specified
    size on the specified virtual node, if possible.  If vnode = 0,
    or if insufficient resources exist on the specified node, attempts
    to allocate the block on an arbitrary node.  If the block cannot
    be created anywhere, returns nil (0).  Otherwise returns a pointer
    to the beginning of the block. */

extern void SHdeallocate ();
/*  char *a;
        Deallocates a block of shared memory previously acquired from
    SHallocate. */
```

2.3. Semaphores.h

```
typedef unsigned Sem;        /* should be opaque */

extern Sem MakeSem ();
/*  unsigned NumUserThreads, initval;
        Creates a semaphore with the specified initial value.
    Returns zero if insufficient resources exist. */

extern void DestroySem ();
/*  Sem semid; */

extern void V ();
/*  Sem semid;
        Increments the value of the semaphore.  If the result is negative or
    zero, unblocks a thread attempting a P operation. */

extern void P ();
/*  Sem semid;
        Decrements the value of the semaphore.  If the result is negative,
    blocks the current thread. */

extern int SemValue ();
/*  Sem semid;
        Returns the value of the semaphore. */
```

2.4. Events.h

```
typedef unsigned Event;      /* should be opaque */

extern Event MakeEvent ();
/*  unsigned VirtualNodeNum;
        Creates an event on the specified virtual node.  Only threads on
    that node can wait or poll for the event, and then only one at a time.
    Any thread can post or destroy the event.  Returns zero if insufficient
    resources. */
```

```
extern void DestroyEvent ();
/*  Event AnEvent; */

extern void PostEvent ();
/*  Event AnEvent;
    unsigned Data; */

extern unsigned WaitEvent ();
/*  Event MyEvent;
        Blocks the caller until the event is posted, then returns the datum
    that was supplied to the post operation. */

extern char /* Boolean */ PollEvent ();
/*  Event AnEvent;
    unsigned *data;
        If the event has already been posted, writes the datum from the post
    operation into the location specified by the data pointer parameter, then
    returns true.  Otherwise returns false.  Does not block. */
```

2.5. DualQueues.h

```
typedef unsigned Queue;      /* should be opaque */

extern Queue MakeDualQueue ();
/*  unsigned numitems;
        Creates a dual queue.  Returns zero if insufficient resources
    exist.  A maximum of numitems data items can be enqueued before
    the queue will overflow.  A maximum of numitems threads can be
    blocked trying to dequeue from the queue when it is empty. */

extern void DestroyDualQueue ();
/*  Queue qid; */

extern void EnqueueDualQueue ();
/*  Queue qid;
    unsigned data; */

extern unsigned DequeueDualQueue ();
/*  Queue qid;
        Blocks the caller until the queue is non-empty, then dequeues. */

extern char /* Boolean */ PollDualQueue ();
/*  Queue qid;
    unsigned *data;
        If the queue is non-empty, dequeues an item and writes it to the
    location specified by the data pointer parameter, then returns true.
    Otherwise returns false.  Does not block. */

extern int DualQueueStatus ();
/*  Queue qid;
        If the queue is non-empty, returns the number of items in it.
```

```
    If the queue is empty, returns the negative of the number of threads
    that are waiting to dequeue data. */
```

2.6. Monitors.h

```
typedef char* Monitor;
/*  actually a pointer to a struct, which should be opaque */

extern Monitor MakeMonitor ();
/*  unsigned NumConditions, unsigned NumUserThreads;
        Creates a monitor with the specified number of conditions.
    Returns nil (0) if insufficient resources exist.  For the purposes
    of WaitCondition, SignalCondition, and ConditionCount, the conditions
    will be numbered from 0 to NumConditions-1.  All queues associated
    with the monitor (entry queue, urgent queue, condition queues)
    will hold a maximum of NumUserThreads each. */

extern void DestroyMonitor ();
/*  Monitor monid; */

extern void EnterMonitor ();
/*  Monitor monid; */

extern void LeaveMonitor ();
/*  Monitor monid; */

extern void WaitCondition ();
/*  Monitor monid;
    unsigned condit; */

extern void SignalCondition ();
/*  Monitor monid;
    unsigned condit;
        Hoare semantics: the signaller relinquishes control of the
    monitor and is placed in an "urgent" queue.  The waiter gains
    control of the monitor with no chance for intervening actions
    by any other thread. */

extern unsigned ConditionCount ();
/*  Monitor monid;
    unsigned condit;
        Returns the number of threads waiting on the condition. */
```

References

[1] BBN Advanced Computers Incorporated, "Chrysalis® Programmers Manual, Version 3.0," Cambridge, MA, 28 April 1987.

[2] BBN Laboratories, "Butterfly® Parallel Processor Overview," BBN Report #6148, Version 1, Cambridge, MA, 6 March 1986.

[3] BBN Laboratories, "The Uniform System Approach to Programming the Butterfly® Parallel Processor," BBN Report #6149, Version 2, Cambridge, MA, 16 June 1986.

[4] C. M. Brown, R. J. Fowler, T. J. LeBlanc, M. L. Scott, M. Srinivas, and others, "DARPA Parallel Architecture Benchmark Study," BPR 13, Computer Science Department, University of Rochester, October 1986.

[5] E. W. Dijkstra, "Co-operating sequential processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, London, 1968.

[6] C. A. R. Hoare, "Monitors: An Operating Systems Structuring Concept," *CACM 17*:10 (October 1974), pp. 549-557.

[7] D. E. Knuth, *Seminumerical Algorithms*, The Art of Computer Programming V. 2, second edition, Addison-Wesley, Reading, MA, 1981.

[8] T. J. LeBlanc, N. M. Gafter, and T. Ohkami, "SMP: A Message-Based Programming Environment for the BBN Butterfly," BPR 8, Computer Science Department, University of Rochester, July 1986.

[9] T. J. LeBlanc and S. Jain, "Crowd Control: Coordinating Processes in Parallel," *Proceedings of the 1987 International Conference on Parallel Processing*, 17-21 August 1987, pp. 81-84.

[10] T. J. LeBlanc, "Structured Message Passing on a Shared-Memory Multiprocessor," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, Jan 1988.

[11] J. G. Mitchell, W. Maybury, R. Sweet, and J. R. Herz, Jr., "Mesa Language Manual, version 11.0," Xerox Office Systems Division, June 1984.

[12] T. J. Olson, "Modula-2 on the BBN Butterfly Parallel Processor," BPR 4, Computer Science Department, University of Rochester, January 1986.

[13] M. L. Scott, "LYNX Reference Manual," BPR 7, Computer Science Department, University of Rochester, August 1986 (revised).

[14] M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering SE-13*:1 (January 1987), pp. 88-103.