SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 26(3), 327–356 (3 1996)

Ariadne: Architecture of a Portable Threads system supporting Thread Migration [*]

EDWARD MASCARENHAS
VERNON REGO
Department of Computer Sciences, Purdue University, West Lafayette, IN 47907.
{*edm,rego*}*@cs.purdue.edu*

SUMMARY

Threads exhibit a simply expressed and powerful form of concurrency, easily exploitable in applications that run on both uni- and multi-processors, shared- and distributed-memory systems. This paper presents the design and implementation of Ariadne: a layered, C-based software architecture for multi-threaded distributed computing on a variety of platforms. Ariadne is a portable user-space threads system that runs on shared- and distributed-memory multiprocessors. Thread-migration is supported at the application level in homogeneous environments (e.g., networks of SPARCs and Sequent Symmetrys, Intel hypercubes). Threads may migrate between processes to access remote data, preserving locality of reference for computations with a dynamic data space. Ariadne can be tuned to specific applications through a customization layer. Support is provided for scheduling via a built-in or application-specific scheduler, and interfacing with any communications library. Ariadne currently runs on the SPARC (SunOS 4.x and SunOS 5.x), Sequent Symmetry, Intel i860, Silicon Graphics workstation (IRIX), and IBM RS/6000 environments. We present simple performance benchmarks comparing Ariadne to threads libraries in the SunOS 4.x and SunOS 5.x systems.

KEY WORDS thread process migration scheduling parallel distributed

INTRODUCTION

In traditional operating systems a *process* is an environment in which a program executes, providing areas for instructions and data. For example, a Unix process typically executes within a single thread of control, with direct access to user data in its own address space and access to system data via system calls [1]. Interprocess or shared-memory communication may be used to provide data-sharing between two or more processes, allowing for either truly concurrent operation on a multiprocessor, or pseudo-concurrent operation on a uniprocessor. The resulting computation may be viewed as multithreaded, with two or more threads of control. Unfortunately, implementing multithreading via processes can be expensive: overheads for interprocess communication, synchronization and context switching may be too high to warrant the effort.

It is possible to implement multithreading within a process, to eliminate the costs incurred by shared-memory calls and communication between processes. Multiple threads of control can be made to share a single address space and run concurrently, as distinct processes. Each such

[*] Research supported in part by ONR-9310233, ARO-93G0045 and NATO-CRG900108.

CCC 0038–0644/96/030327–30
©1996 by John Wiley & Sons, Ltd.
Received 9 February 1995
Revised 23 July 1995

thread within a process may execute as a lightweight process (LWP), with a program counter and a stack for maintaining local variables and return addresses, and with free access to process variables. By keeping thread contexts small, context-switching costs between threads can be made a fraction of context-switching costs between processes. Many threads may coexist in a process's address space, allowing for cheap and efficient multithreading. Because threads may execute concurrently, however, programming with threads requires care [2]. For example, because a thread is free to access any virtual address, it can access and (inadvertently) destroy the stack of any other thread [3].

Ariadne is a threads library that provides lightweight threads in multiprocessor Unix environments. A single process may host thousands of threads, with low cost thread-creation, context-switching, and synchronization. The library is highly portable: nearly all of the code is written in C, with the exception of hardware dependent thread initialization. Ariadne has successfully been ported to the SPARC (SunOS 4.x and SunOS 5.x), Sequent Symmetry, IBM RS/6000, Silicon Graphics workstation (IRIX 5.2), and Intel iPSC environments. Support for parallel programming on shared-memory multiprocessors and in distributed-memory environments is provided via inter-process thread migration. Distributed-Ariadne is realizable through any one of a variety of communication libraries or parallel programming environments, because thread-migration is layered upon communication primitives. Current implementations support PVM [4] and Conch [5], though Ariadne's communications interface makes these easily replaceable by other communication libraries.

Our original motivation for designing and implementing the Ariadne system was to support process-oriented parallel and distributed simulations [6]. Process-based simulations are popular because they are known to map easily into implementations [7]. The idea is to identify active objects in a real system and implement them as processes in a program which simulates the system. Though we are not aware of simulation systems, in particular parallel systems, that have previously used threads in this manner, the lightweight nature of threads – affording inexpensive communication, switching, and data sharing – make them natural candidates for implementing active simulation objects.

Implementing process-oriented simulations on parallel and distributed systems entails remote data access. A thread executing within a process on one host may need access to data available on a process residing on another host. Consider, for example, an active object such as a "customer" (e.g., a packet in a data network) which executes on one host and requires access to a passive object such as a "server" (e.g., a gateway in a network) located on a remote host. This is solved in one of two ways: either move the requested data to the host on which the thread is located or move the requesting thread to the host on which the data is located. In most simulation applications, however, passive objects are sufficiently large (e.g., servers are usually associated with queues, involving possibly complicated data structures containing transient active objects) to discourage frequent data migration. On the other hand, threads representing active simulation objects are generally small. In our experience, the cost of moving thread state from one processor to another – thread-migration – is equivalent to the cost of passing messages containing the same event-related information [6].

Operating in concert with our motivation for implementing processes in parallel simulations are our requirements of portability and flexibility. Portability is important to us because of the cheap parallelism available on workstation clusters and hardware multiprocessors. We exploit Unix library primitives such as `setjmp()` and `longjmp()` to minimize hardware dependencies in the implementation of context-switching. Despite this, a small portion of thread initialization must be coded in assembler. While Ariadne provides an efficient internal scheduler, it is flexible in that it also provides for customized schedulers. The internal scheduling policy is based on priority queues: at any given time, the highest priority non-blocked thread runs. A customizable scheduler is provided simply because different applications may require different scheduling policies. For example, in simulation applications, threads are scheduled based on minimum time-stamp.

Though this was not a guiding design consideration, we have found Ariadne to be useful in general multithreaded distributed computing [8]. It has been used for parallel applications requiring a large number of threads: particle physics, numerical analysis, and simulation. The choice of granularity, or equivalently, the mapping between a thread and a set of system objects is user-dependent. For example, in particle-physics applications a thread may represent one or more particles; in Successive Over-Relaxation (SOR) methods [9] for solving systems of linear equations, a thread may represent computation at one or more neighboring grid points on a template; in simulations of personal communication systems a thread may represent a mobile phone. Dynamic system objects are naturally representable by mobile threads: computations which can migrate between processes. These applications suggest that Ariadne may be used for a variety of parallel and distributed computations.

Related Work

Threads can be supported in user-space or in kernel-space. Each approach has its advantages and disadvantages. Some examples of user-space threads systems include the Sun-LWP lightweight process library on SunOS 4.x [10], Quick Threads [11], an implementation of the POSIX standard for threads [12], Rex [13], and Cthreads [14]. Because kernel support is not required for implementing user-space threads, they can be implemented on an OS that does not provide threads. User-space threads are relatively easy to implement, and can be made portable and flexible; they can be modified without making changes to the OS kernel. Context-switching between user-space threads is at least an order or magnitude faster than kernel traps [3]. Other advantages include customizability and scalability. The latter is particularly important because space for tables and stacks can be hard to come by in the kernel. Unfortunately, user-space threads can perform poorly in the presence of I/O calls, page faults or system calls: when a thread blocks, its host process is blocked until the I/O is done, a page is loaded, or the requested OS service is granted.

In kernel-space implementations, threads are created and destroyed through kernel calls, with no need for a runtime system [3]. Thread-state information is kept within the kernel, and is associated with a thread-entry in a process-table. Calls that may block a thread are implemented as system calls – implying a higher cost than with a corresponding call to a runtime system in user-space threads. When a thread blocks, the kernel may switch control to a runnable thread within the same process, if one exists, or switch control to a thread within another process. A key advantage of kernel-space threads is that threads which block do not prevent other threads from running.

Because kernel-threads can impair performance when present in large numbers, it is usual practice to create only as many kernel-threads as the attainable level of concurrency, given an application and execution environment. Multiprocessor operating systems such as SunOS 5.3 [15], Mach [16], and Topaz [17] provide support for kernel threads. Since kernel-thread management primitives perform poorly in comparison to user-space threads, it is not uncommon to have user-space threads created atop kernel-threads. Systems exemplifying this approach include the multithreads library on SunOS 5.x (Sun-MT [18]), Mach(CThreads), and Topaz(WorkCrews [19]). User-threads are multiplexed over available kernel-threads in a shared-memory multiprocessor environment.

An idea advocated recently is the combination of kernel-space and user-space threads, to obtain the advantages of both. Scheduler activations [20], and Psyche [21] are examples of such systems. In scheduler-activations, kernel-threads do up-calls into user space providing execution contexts that may be scheduled on the threads system. When a scheduler activation is about to block, an up-call informs the threads system of this event, enabling the system to schedule another thread. In Psyche, the kernel is augmented with software interrupts which notify the user of kernel events. The kernel and the threads system share data structures so they can communicate efficiently. To implement these mechanisms, it is necessary to modify parts of the OS kernel.

Ariadne is a user-space threads library for parallel and distributed systems. On shared-memory multiprocessors, Ariadne multiplexes user threads on multiple processes, as is done in PRESTO [22]. An Ariadne thread accesses a remote object in a distributed-memory system by migrating to the location of the object. Other systems providing parallel programming with threads include Amber [23] and Clouds [24]. As in Ariadne, these systems provide threads and objects as basic building blocks. By providing a neat interface between the threads library and a parallel programming environment, with requisite support for multithreading, Ariadne encourages tailored solutions to problems. In this paper we describe the architecture of Ariadne: the focus is on design, implementation, and performance. Customization features with the parallel programming model, and detailed examples, are provided in Reference 8.

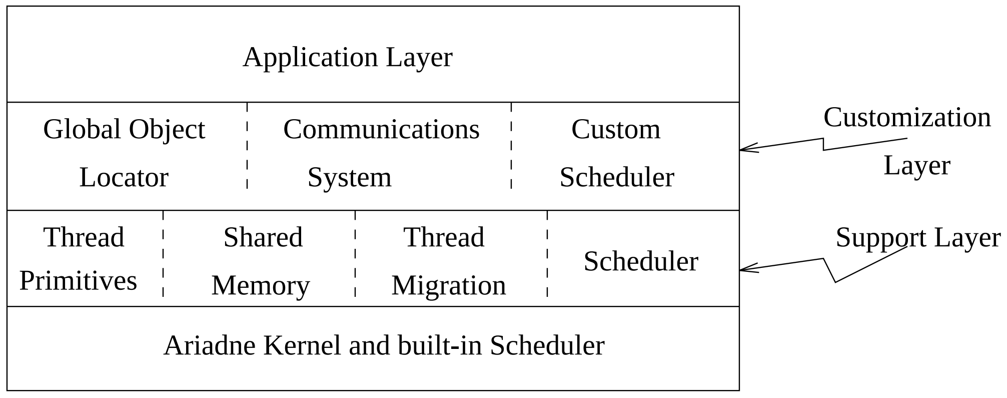

Figure 1. Ariadne Architecture

OVERVIEW OF ARIADNE

The Ariadne system consists of three layers, as shown in Figure 1. The lower-most layer contains the kernel, the middle layer provides customization and thread support, and the top layer provides custom modules. The **kernel layer** provides facilities for thread creation, initialization, destruction, and context-switching. The **support layer** enables end-applications and software systems to use Ariadne via supervised kernel access. The **customization layer** provides a set of independent components that aid in customizing Ariadne for specialized use; e.g., the use of Ariadne for simulation or real-time applications. The Sol [25] simulation library uses Ariadne for scheduling and handling events – the next thread to run is a handler for the lowest time-stamped event in the system.

The kernel is interfaced with a built-in scheduler based on priority queues. Each Ariadne thread is assigned an integer priority ranging from `1` to a user-specified `N`. At any given time, a highest priority runnable thread executes. Through basic primitives provided in the support layer, Ariadne can be tailored to offer thread migration, specialized schedulers, and multithreaded computations on shared- and distributed- memory machines. When a customized scheduler is used, Ariadne's kernel invokes user functions (up-calls) to fetch the next runnable thread; if a running thread blocks or suspends execution within an Ariadne primitive, the kernel returns the thread to the user via another function. How the customization layer is used depends on the needs of the application. For transparent access to global objects, object-locator software may be used. For distributed computing, PVM, Conch or equivalent communication software may be used.

Design Considerations

We now elaborate on some of the design goals mentioned briefly in the introduction. Any evaluation of the system is best viewed in terms of these goals.

- **Portability.** Ariadne is targeted towards Unix-based machines. Apart from assembly-level thread initialization code, the entire system is written in C. Thread initialization involves allocation of a new stack and register loading, so a thread can begin execution when given control. The stack pointer and the program counter must be loaded, and the stack initialized with an initial frame. Context-switching involves saving a running thread's register state, including all floating point registers, and the signal mask (if required), and restoring the next runnable thread's corresponding state. For portability, Ariadne accomplishes all this using the standard `setjmp()` and `longjmp()` subroutines* available in Unix environments [1]. Porting Ariadne entails rewriting only thread initialization code – about 5 lines of assembly code on a SPARC!
- **Ease of Use**. The basic user interface is simple. Use of Ariadne requires a basic understanding of concurrent programming and C/C++. There is no limit placed on the number of threads usable at a given time, allowing for a natural representation of parallelism intrinsic to many applications. A thread may migrate from one process to another, at any point in its execution sequence, via a function call. Transparent thread migration and transparent scheduling of threads on available processors makes for a simplified parallel programming interface.
- **Customization**. Though primarily designed for simulation, Ariadne can support a variety of applications. The customization support layer allows Ariadne to be tailored to suit the needs of a specific application. At present this layer supports customized schedulers, arbitrary communication libraries, and object-locator software. For example, Distributed-Ariadne on the Intel iPSC can make use of the native communications library on that system. Customization has been used to implement a time-stamp based scheduler for simulation, and a shared-memory based scheduler for use on shared-memory architectures. An example of a simple object-locator for a particle-physics application is described in the Section on Object Location.

* Our portability requirements have been met with this approach. If particular implementations of `setjmp()` and `longjmp()` violate our assumptions, these routines may be replaced.

- **Efficiency**. Ariadne was designed to be streamlined and efficient, though portability and simplicity considerations overruled performance optimizations. Efficiency measures were adopted whenever these did not conflict with requirements of portability and simplicity. For example, assembly-level implementation of context-switching outperforms our current implementation. But performance measurements described in a later Section indicate that the difference is not significant. For efficiency, Ariadne implements context-switching via a scheduling function instead of a thread. Though this introduces race conditions and complicates the implementation, such complexity is transparent to the user. Details on how race conditions are handled are provided in the Section on Shared-Memory Multiprocessor Support.

A Simple Example

We illustrate the use of Ariadne with a simple example on sorting, using the well-known quicksort algorithm [26]. As shown below, the sequential `QUICKSORT(S)` function operates recursively, to sort a list S of n integers.

```
QUICKSORT(S)
{
  if ( S contains at most one element )
    return S;
  else {
    let elem = a pivot element chosen randomly from S;
    let S1 be the set of elements < elem;
    let S2 be the set of elements = elem;
    let S3 be the set of elements > elem;
    return QUICKSORT(S1);
    return S2;
    return QUICKSORT(S3);
  }
}
```

To sort an array of n integers, quicksort uses a `partition()` procedure which chooses a pivot to divide the array into three parts – the first part containing elements with value less than the pivot, the second part containing elements with value equal to the pivot, and the third part containing elements with value greater than the pivot. The quicksort routine is then invoked recursively to operate on two parts.

An Ariadne program which implements the quicksort on a shared-memory multiprocessor is shown in Figure 2. The conversion from a sequential program to a parallel program is simple and natural: the sequential quicksort routine turns into a thread. Function `main()` sets up the shared environment using primitives `a_set_shared()` and `a_shared_begin()`. Function `create_scheduler()` implements a special scheduler for shared memory. An example of Ariadne's scheduler customization feature, this scheduler allows different processes to obtain runnable threads from a common queue.

Function `ariadne()` initializes the threads system, and function `main()` – which now becomes a thread executing at priority 7 – creates a single `quicksort()` thread and exits. In truth, the main function suspends its own execution within `a_exit()` until all other threads are done. New threads are created using function `a_create()`. Following the main

```
#include "aria.h"
#include "shm.h"
void quicksort(int *ia, int low, int high)
{
   int index;   /* the partitioning index */
   if (low < high) {
      /* use first element to partition
      partition the array and return the partitioning index */
      partition(ia, low, high, &index);
      /* create a new thread to sort S1, the smaller part */
      a_create(0, quicksort, 6, SMALL, 3, 0, 0, ia, low, index-1);
      /* S3, the larger part continues to be sorted by the calling thread */
      quicksort(ia, index+1, high);
   }
}

#define SHM_KEY 100
int *aa;
main(int argc, char *argv[])
{
   struct thread_t whoami;   /* main thread identifier */
   int size;   /* size of array to be sorted */
   int shm_id;   /* identifier for shared segment */
   int nprocs;   /* number of processes to run */
   clock_t msecs;

   /* input the value of nprocs and size */
   /* set up the shared environment */
   a_set_shared(0);
   a_shared_begin(nprocs);
   create_scheduler();
   /* create a shared segment to hold the array */
   aa = shmcreate(SHM_KEY, sizeof(int)*size, PERMS, &shm_id);
   ariadne(&whoami, 7, 1024);
   if (sh_process_id == 0) {   /* parent process ? */
      srand(12345);   /* initialize seed */
      for (int i=0; i < size; ++i)
         *(aa+i) = rand();   /* init array to be sorted */
      a_create(0, quicksort, 6, SMALL, 3, 0, 0, aa, 0, size-1);
   }
   msecs = clock();
   a_exit();
   msecs = clock() - msecs;   /* for timing */
   /* print results */
   a_shared_exit();
   shmfree(aa, shm_id);
}
```

Figure 2. Quicksort Program in Ariadne

function's exit, the newly created `quicksort()` thread – executing at priority 6 – begins to run. It chooses a pivot to split the array into three parts, creating yet another (child) thread to sort elements that are smaller than the pivot. It continues to run as a parent thread, making a recursive call to quicksort, to sort elements larger than the pivot. In this way, a system of threads is created to work on sorting distinct pieces of the array. When all quicksort threads have completed their work and exited the system, the main thread returns from `a_exit()` and may output the sorted array.

This program runs on any supported architecture without modifications. Load balancing is automatic – a single queue stores work to be done in the form of runnable threads. When a process becomes free to do work it acquires a thread from the head of the queue. Though threads may be created by one process, they may be run by another process. Thread-migration between processes is transparent to the application. It is not necessary to use Ariadne's synchronization mechanisms in this example because each thread works on a distinct piece of the array.

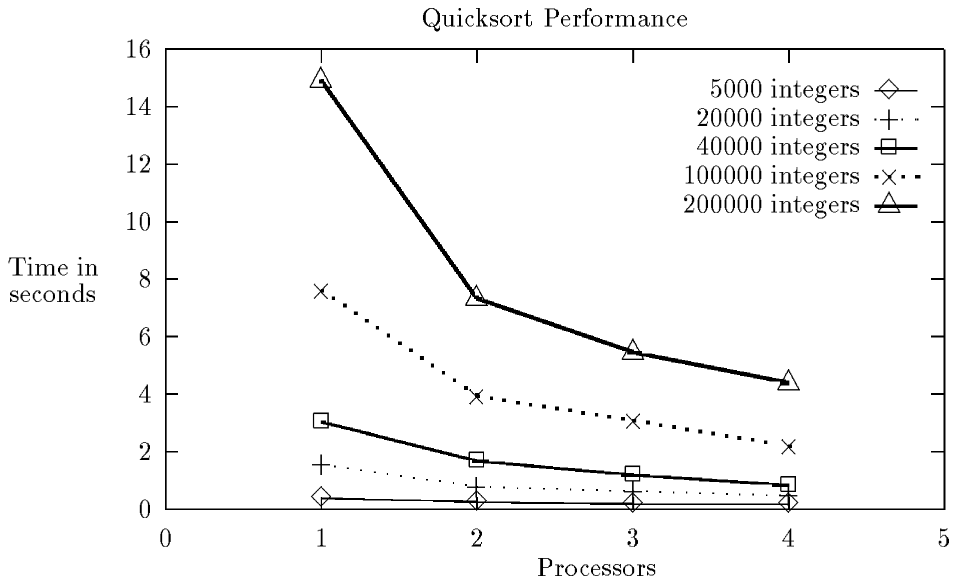

Figure 3. Quicksort performance: shared-memory Ariadne

Multithreaded quicksort performance

The performance of the multithreaded quicksort, on integer arrays ranging from size 5000 – 200000, is shown in Figure 3. All measurements were made on a 4-processor SPARCstation 20. Initialization time is ignored, and only the (user + system) time required for sorting is reported. Because of the additional overheads imposed by thread operations, small arrays are more effectively sorted without creating and executing new threads. For example, the cost of thread creation is roughly equal to the cost of a 200-integer bubblesort on the SPARCstation

20. For this reason, when an array partition reaches size 200, the quicksort routine in Figure 2 may simply invoke a bubblesort procedure instead of creating a new quicksort thread. This approach was taken for the performance measures reported here.

As shown in Figure 3, good speedups are obtained in sorting large arrays. For 5000-integer sorts, the curve flattens out at 2 processors; for 200000-integer sorts, speedup appears likely even with more than 4 processors. The total number of threads created depends on the size of the initial array: from 75 threads for the 5000-integer array to 2981 threads for the 200000-integer array. The maximum number of active threads coexisting at any given time ranged from 8 to 240, respectively, for these two extreme cases. Multiprocessing load was balanced because the number of thread creations and context-switches within each process were roughly the same. In general, performance depends on choice of the pivot element.

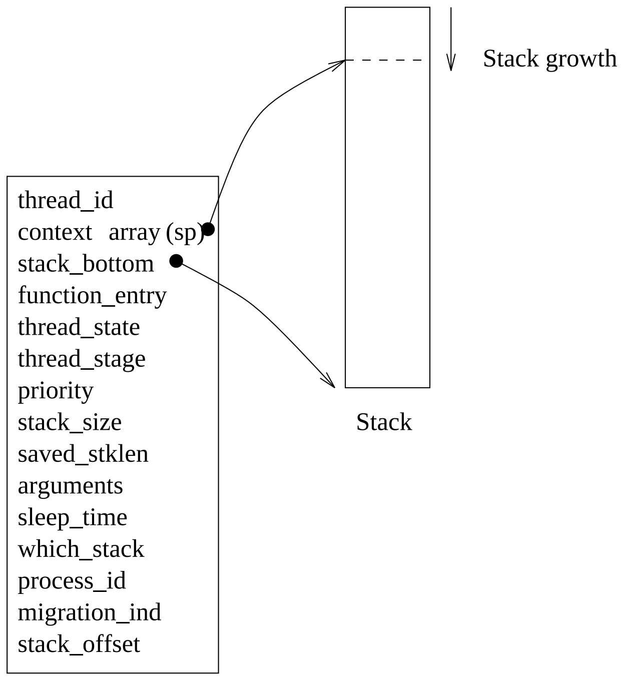

Figure 4. Thread Context Area (`tca`)

Basic Kernel Mechanisms

A large number of threads may run within a single Unix process. Each thread's state information is kept inside a thread *shell*: a thread context area (`tca`) that is similar in appearance to the process control block an OS maintains for each process, along with stack space. The layout is depicted in Figure 4. Each thread is labeled with an identifier that is unique to its host process. When combined with the `id` of the process at which it is created, this identifier is also unique to a system of processes. Depending on user specification, each thread may either run on its own stack or on a *common stack* shared by all threads. The `tca` contains a context array for saving registers. The size of this array depends on the size of `setjmp`'s buffer, as declared in `setjmp.h`. Before relinquishing control to another thread, a thread must first save its state in this array. Before a thread is run, its state must first be restored from this array. Also stored in the context array is a stack pointer (`SP`). Other information stored in the `tca` include the function entry – the function the thread is to execute, an argument list, thread priority, thread status, and stack-related data.

Thread Creation and Destruction

A thread is created from a thread shell by specifying the thread's function, priority, stack size, and function parameters. Function `a_create()` attempts to obtain a shell from a free list of shells containing stacks of the specified size. If the list is empty, a new shell is created. Next, the specified function is bound to the thread and the thread is placed on a ready queue of runnable threads. Which queue is chosen depends on the specified priority of the newly created thread. A thread may be destroyed when it no longer needs to run, or when it has run to completion. To enhance performance, the shell (i.e., `tca` and stack) of a dead thread is recycled: it is kept on a free list for reuse by another thread with a similar stack requirement. New thread shells are created only when no recycled shells are available. Threads may be given a variable number of integer, float, and double type arguments. These are kept in temporary store within the `tca`, until the thread begins to run.

Stack Allocation

Ariadne threads may request stacks of any size. One possibility is to use encoding, where the user defines the size of `TINY` stacks in Ariadne's initialization call. After initialization, the user may create threads with stacks of size `SMALL`, `MEDIUM`, `LARGE` and `HUGE`, each of which is defined relative to size `TINY`. Alternately, specific stack sizes may be defined in each `a_create()` call. For efficiency, stacks are allocated using a lazy-binding scheme. That is, a stack is (permanently) bound to a thread shell only just before the thread begins to run for the first time. As a result, only threads which actually run – and not all threads created – use up valuable stack space.

Despite the precautionary lazy-binding scheme, an acute memory shortage may arise if each of many coexisting threads asks to run on a large stack of its own. To solve this problem, Ariadne lets threads use a large *common stack* allocated in static memory. All threads with large dynamic memory requirements may ask to run on the common stack. On a context-switch, a thread's essential stack (i.e., occupied portion of stack) is saved in a private area. In most applications, the size of the essential stack is smaller than the size of the stack requested. Though a shared stack reduces overall memory requirements, the cost of saving/restoring stack contents on each context-switch imposes a small additional overhead on the threads system.

On each context-switch, Ariadne checks for stack overflow. This is done by testing for the presence of a "magic number", initially placed at the far end of the stack during thread creation. While this technique is effective, it does not guarantee detection of overflow. Because the magic number may not be overwritten on all stack overflows, Ariadne delivers a warning when stack usage approaches a predefined fraction of the total stack size.

```
/*
* Assume "curr" and "next" contain pointers to the tcas of the
* currently running thread and the next thread to run, respectively.
*/
void ctxsw(void)
{
   ...
   if (!setjmp(curr->context)) {     /* saves registers */
      old = curr;
      curr = next;
      curr->state = RUN;
      if (curr->stage == UNBORN)
         init(curr);     /* first time initialization */
      else
         longjmp(curr->context, 1);     /* restores registers */
   }
   old->state = READY;
   ...
}
```

Figure 5. Context Switch

Context-Switching Mechanism

A thread that is currently running is called the *current* thread. Context-switching – triggered by functions `a_resched()` and `a_yield()` – is performed on the stack of the current thread when it "yields" or relinquishes control. A segment of code illustrating this idea is shown in Figure 5. During a context-switch, the current thread is placed at the tail-end of its priority queue, but is not marked *READY* (runnable) until the switch is complete. Because the switch is performed on the current thread's stack, marking it *READY* before the switch is complete may create race conditions on a multiprocessor. When the switch is complete, the *next* thread to run – the "target" selected by the scheduler – is redefined to be the current thread, and the thread that just lost its status as current thread is marked *READY*. The states of the current (yielding) and next (target) thread stacks are shown in Figure 6.

As indicated by the code shown in Figure 5, a `setjmp()` saves the yielding thread's context in the `tca`, and returns a value of 0. An ensuing `longjmp()` uses a given `tca` to make the corresponding target thread resume execution at the point where it last invoked `setjmp()`, and returns a value of 1. This return value is used to differentiate between a context-save and a context-restore, so that an appropriate course of action can be taken. On some architectures (e.g., IBM RS/6000), the `setjmp()` and `longjmp()` primitives prohibit jumps from lower to higher addresses (or vice versa, if the direction of stack growth is reversed), necessitating some hand-coding. Note that context-switching activity is independent of scheduling activity; the scheduler's job is only to determine the identity of the target thread and invoke context-switching code. For efficiency, Ariadne inlines context-switching code for the generic context switch, yield, and migrate primitives.

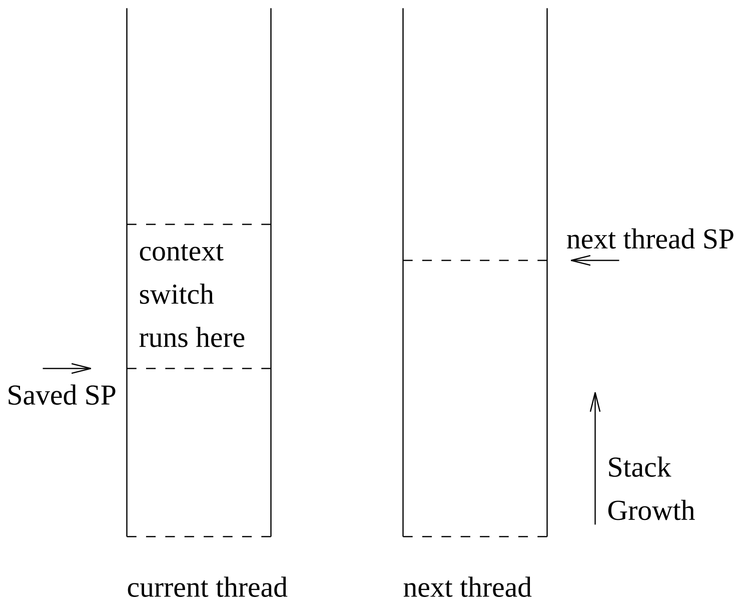

Figure 6. State of stacks during a context switch

Scheduling

The Ariadne system provides a built-in scheduler that allocates portions of a host process's time-slice to its resident threads. At any given time, the highest priority runnable thread runs. Within a priority class scheduling is FIFO. An executing thread continues to run until it terminates, completes a time-slice (if the system is in time-slicing mode), or suspends execution.

Allowing threads at the same priority to share a CPU via time-slicing is useful. Consider an application that uses one thread for computing the value of a function, and another thread for writing to a file. Time-slicing offers the potential for multiplexing between I/O and computations, or between distinct types of computations. Time-slicing is accomplished using the signal mechanism, through Unix SIGALRM and SIGVTALRM signals, i.e., execution-time or elapsed-time based. Slice granularity can be set using the `a_defslice()` primitive.

Critical Sections

When a thread completes its execution slice, or when a sleeping thread awakens, Ariadne's interrupt handlers get control. Such interrupts affect control flow in the threads system. If a thread is in a critical section, manipulating Ariadne's internal data structures, it is crucial that the thread it be allowed to operate without interruption. To enable this, Ariadne uses a flag to protect critical sections. If the flag is found set when an interrupt occurs, the interrupt handler returns control to the thread without handling the interrupt. Such handling is resumed when it is safe to do so. A thread that is interrupted in a critical section is allowed to run for an additional time-slice, with the hope that it may exit the critical section. Masking interrupts before entering a critical section is also possible, but involves an expensive Unix system call.

Ariadne provides users with `a_interruptoff()` and `a_interrupton()` primitives, based on masking mechanisms, for protecting critical sections in applications. They are also useful in preventing interrupts from occurring during non-reentrant C library calls, e.g. `printf()`. If an interrupt occurs while a thread is within a critical section, the interrupt is held and delivered to the thread when the critical section is exited. Time-slicing with masked interrupts allows for a fair distribution of execution slices among runnable threads.

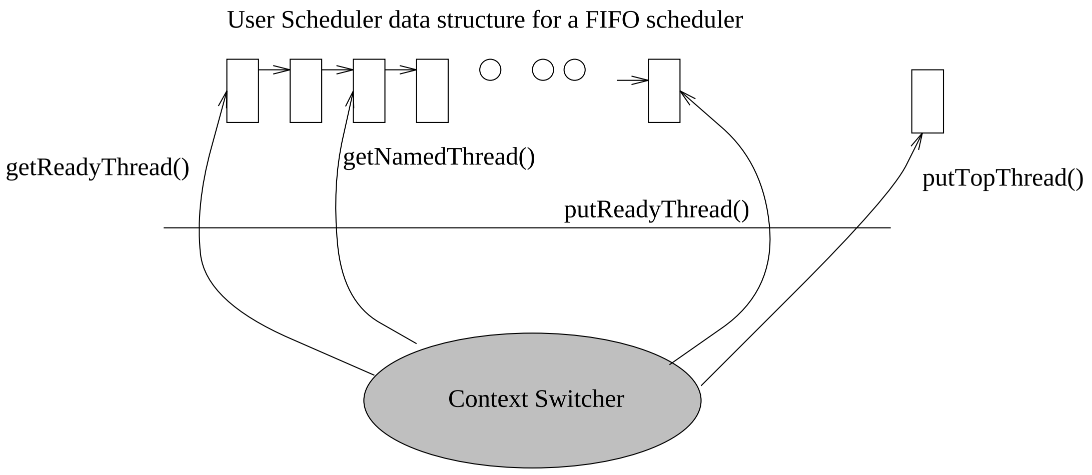

Figure 7. Scheduler Customization

Scheduler Customization

Ariadne was designed to support parallel and distributed simulation applications where threads operate as processes that simulate active-system entities. In such a framework, threads are scheduled for execution based on minimum time-stamp. Each thread may be viewed as a handler for a sequence of events. It runs to perform simulation work when some scheduled event it is associated with occurs. Because Ariadne's internal scheduler selects a next thread to run based on highest integer priority, it does not directly support time-based scheduling or more general forms of scheduling. To enable flexible scheduling, Ariadne provides an interface for the development of customized schedulers. For example, imagine using a thread to recursively compute $n!$. This thread recursively creates a child thread to compute $(n-1)!$, and the child creates yet another thread to compute $(n-2)!$, etc. The value returned by the nth thread is the product of n and the value returned by its child, for $n \geq 1$. For such a computation to work, the thread computing the base value of the recursion (i.e., 0 or 1) must be the first to execute. This requires a scheduler which schedules threads for execution in reverse of the order in which they are created (i.e., LIFO).

A customized scheduler may be installed using the `a_create_scheduler()` primitive.

For example, the schematic for a FIFO scheduler is shown in Figure 7. The user must provide four functions that are callable by the Ariadne kernel during a context switch. The user must also maintain the `tca`s of runnable threads in an appropriate data structure (e.g., a linked list). A `getReadyThread()` primitive extracts the next thread to be run (located at the head of the list in the figure) from the data structure and returns it to Ariadne's kernel, which then gives control to the thread. The `putReadyThread()` primitive returns a yielding thread to the user. The yielding thread must be inserted into an appropriate position in the data structure, so that it may subsequently be retrieved for execution when `getReadyThread()` is invoked. What these functions do depends on the scheduling policy required by the application. In Figure 7, `putReadyThread()` places the thread at the tail of the list.

A thread returned to the user by `putReadyThread()` may have terminated execution. The user may reclaim its shell (`tca` and stack) for reuse or simply allow the Ariadne kernel to reclaim it on its free lists. The two remaining primitives shown in the figure may be used in special situations. If the Ariadne kernel needs to temporarily interrupt and suspend execution of a thread while some important internal function (e.g., check-pointing of thread state) is to be executed by another thread, the kernel returns the former thread to user-space by invoking `putTopThread()`. The user may then return this thread to the scheduler on the next call to `getReadyThread()`, if necessary. The kernel uses `getNamedThread()` to obtain a specific thread from the user. To enable this, the data structure used must support removal of a specific thread based on a key.

SHARED-MEMORY MULTIPROCESSOR SUPPORT

Ariadne provides support for the concurrent execution of threads on shared-memory multiprocessors. Its portability requirements currently preclude kernel level support from a host's OS. Nevertheless, Ariadne exploits multiprocessing power by multiplexing threads on distinct processes, generally using as many processes as available processors. Besides requiring process creation, there is an added disadvantage: when a thread blocks on a system call, it forces its host process to also block, effectively blocking runnable threads*. On shared-memory multiprocessors, Ariadne's initialization call forces the `main()` function to fork off a user-specified number of processes. Each process is assigned a unique integer identifier `sh_process_id` and granted access to shared memory segments. The parent process is assigned the identifier zero (see Figure 8), and the child processes are assigned identifiers 1, 2, 3, etc. These identifiers permit processes to selectively execute parts of user code.

The basic idea is to allow the shared memory multiprocessor host a number of Unix processes, each of which hosts a number of Ariadne threads. The threads interact via Ariadne primitives which in turn operate on shared-memory. Each process maintains a private stack and notion of a "main" thread, not shareable with other processes. At any given time, each process may run a distinct thread, independently of the other processes. Shared-memory synchronization constraints may cause dependencies between thread-execution sequences on distinct processes. A number of Ariadne's internal (global) state variables that are private on a uniprocessor must become shared variables on a multiprocessor. Examples of such variables include the live-thread count (which aids in system termination), the priority of the highest-priority runnable thread, and the identifier of the most recently created thread (which is useful in identifier allocation). Storage of these and other user-specified shared variables is made possible through creation of a primary 1 MB (default) shared memory segment; a user may

* Implementation of an Ariadne interface to kernel-space threads is in progress, to solve this problem on systems supporting kernel threads.

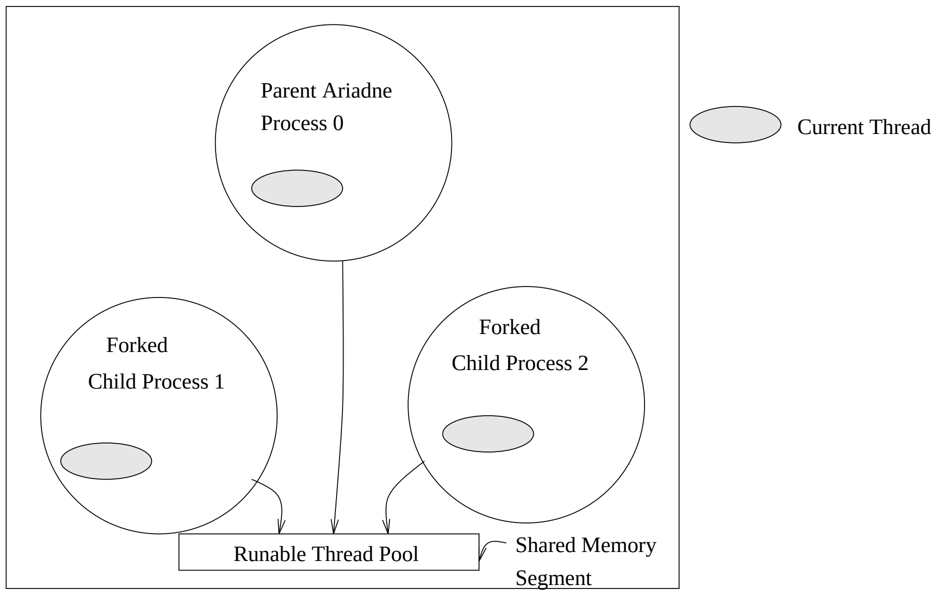

Figure 8. Shared-memory multiprocessing with Ariadne

create/free additional shared memory segments with the `shmcreate()` and `shmfree()` primitives. Dynamic shared memory areas can be obtained from the primary segment for user data via `shmemalloc()` and `shmemfree()`. These are based on IPC facilities available on Unix System V. Memory management within the primary shared segment is handled inside Ariadne, and is transparent at the user level. Internally, Ariadne uses this area to allocate thread shells at runtime. Examples demonstrating use of these primitives are given in the the Performance Results and Examples Section.

Thread Scheduling

Each Ariadne process executes as a user process on a multiprocessor. The OS schedules processes for execution on arbitrary processors as they become available. A problem* arises in the following situation: the OS preempts a process holding a resource (e.g., its OS time-slice may have expired) and schedules a process with a blocked thread – one that is waiting for a thread in the preempted process to release the resource it holds. Since the blocked thread cannot run until the resource is released, at least one cycle is wasted. When the preempted process finally obtains a time-slice, the resource may be released and the process hosting the blocked thread may run on its next attempt.

Data structures for Ariadne's scheduler can be placed in shared memory, so that processes can compete for access to the next runnable thread. The shared structure must be locked, accessed and then unlocked, to prevent simultaneous accesses by two or more processes. The scheduler customization feature described earlier may be used in implementing such a

* See last footnote.

scheduler. Two scheduling policies based on this approach have been implemented. In one, the scheduler scans the ready queue, moving in the direction of decreasing priority. The first ready thread found is run. With this, it is possible for threads at different priority levels to execute simultaneously on different processors. An example illustrating the use of this scheduler is given in the Performance Results and Examples Section. In the other scheduler implementation, all processes must execute threads belonging to what is currently the highest priority class. The priority of this class is j if at least one runnable thread of priority j exists, and all other runnable threads have priority i, with $0 \leq i \leq j$. At an extreme, this policy may cause all but one Ariadne process to idle, while the busy process executes a highest-priority thread. Runnable threads at lower priorities have no recourse but to wait for the system's definition of highest-priority to reach their level. A process will not run a thread whose priority is lower than the priority of a thread that is being run by another process.

Race Conditions

On a shared-memory multiprocessor, different processes progress at different rates and execution sequences are nondeterministic. Because of this, race conditions are an inherent part of concurrent thread execution. To make execution sequences meaningful, protection against race conditions is necessary. For example, thread A must be protected from *yielding* control to or destroying thread B, if the latter is already running on some processor, or is currently blocked on some request.

Ariadne ensures that race conditions will not arise during a context-switch from one thread to another. On a given processor, there is a finite amount of time between the instants at which execution of one thread is suspended and another begins to run. During this time, both the thread being suspended and the thread scheduled to run next are said to be in states of "transition". When a thread is in a state of transition, all internal Ariadne operations on the thread are forced to complete without interruption. For example, consider the operation involving transfer of control between a yielding and a target thread. While in transition, both the yielding and target threads are prohibited from receiving control from other threads, or being scheduled for execution by other processes. The yielding thread is marked as being in a state of transition and placed on the ready queue. The target thread marks the yielding thread as READY for execution, only after it receives control. Since a number of scheduling-related operations are performed on the stack of the yielder, a simultaneous transfer of control to the yielding thread from another yielder is prohibited until the current transfer is complete.

DISTRIBUTED-MEMORY MULTIPROCESSOR AND NETWORK SUPPORT

Ariadne may be used for multithreaded computations on distributed memory multiprocessors: networks of uniprocessors and/or hardware multiprocessors. With an enhanced architecture and functionality, Ariadne is able to support distributed multithreading at the application level. The distributed environment may be configured according to a user's need, perhaps utilizing a given communications library or distributed computing software. A clean and simple interface allows Ariadne to be used flexibly with arbitrary communications software systems. Prototype implementations of distributed-Ariadne include support for PVM 3.3.4 [4] and Conch [5].

On a distributed-memory system, Ariadne initializes one or more distributed processes – henceforth called D-processes – on each processor. Each D-process is given a unique integer id, starting with 0. Though a D-process may fork off children on a uni- or multiprocessor, it is responsible for handling all communication between itself (and its children) and other hosts in the system. Child processes are identified with an integer-tuple `<process identifier, shared process identifier>`. A graphical depiction of this architecture is shown in Figure 9. Here three D-processes are shown: D-processes 0 and 2 are located on different multiprocessors, and process 1 is located on a uni-processor. On each multiprocessor, the D-process forks off child processes; a parent and its children share a common threads pool. The uniprocessor hosts a single D-process.

Ariadne's D-processes communicate via messages, using primitives from the underlying communications library (PVM, Conch, etc). Child processes interact with their parent D-process via shared-memory. This results in significant cost savings, since shared-memory IPC is cheaper than socket-based IPC. Messages bound for other hosts are placed in a shared message queue. Parent D-processes retrieve messages from this queue and route them to destination D-processes. Ariadne's D-processes may either be dedicated to communication or share computational load with children. This choice depends on the message density of a given application.

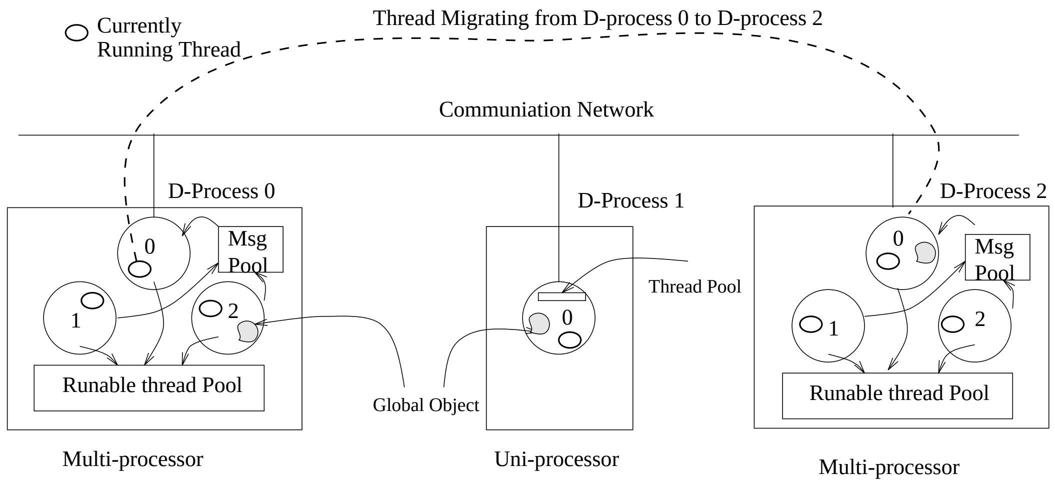

Figure 9. Distributed Ariadne

The Distributed Programming Model

Ariadne's distributed computing model (see Figure 9) has three components: Ariadne processes (circles), Ariadne threads (ellipses), and user objects (shaded areas). Objects may be simple (e.g., integers), or complex data-structures (e.g., lists or trees). Objects may be *global* (i.e., located on the heap or data area, accessible to all threads in a process), or *local* (i.e., located on a threads' stack and private). Ariadne processes encapsulate Ariadne threads and global objects.

Threads are active and mobile computational units, with freedom to move between all Ariadne processes in the distributed environment. In Figure 9 is shown a thread migrating from D-process 0 to D-process 2. Typically, threads move to access global objects located at other Ariadne processes – as computations that chase data. Such objects are static, and there is generally no motivation to migrate them. For thread-migration, Ariadne depends on the use of an object locator: a migrating thread needs to know which host it must migrate to in order to access required data.

A migrating thread takes all its local objects along with it on its travels. Because some local objects may be pointers, post-migration processing may be required to interpret stack addresses correctly. Pointers that point to other local objects must be "updated" at destination hosts. The `a_updatep()` primitive is provided for this purpose. Pointers that point to global objects may also be updated, provided a user-specified update method is available at the destination. For example, a global object identifier may be passed on a thread's stack to aid in this update. At the destination, the application's update function may use the identifier on the stack to locate a copy, replacement, or equivalent object pointer. The application must make such an object available at a destination processor.

Thread migration has been shown [6] to offer definite advantages in parallel simulation and other applications. Advantages include program simplicity, locality of reference, and one-time transmission. The Distributed Shared Memory (DSM) approach for accessing objects requires a remote procedure call (RPC) like mechanism, with round-trip transmission delay [6]. DSM algorithms require some combination of data replication and/or data migration. Which method works best for remote object access depends on the application and the execution environment. A combination of methods may also work well. If a thread repeatedly accesses large objects that are situated remotely, it makes sense for the thread to migrate to the process hosting the object. Data migration will be poor if data size is large or replicated write-shared data is used. Thread migration will perform poorly if a thread's state is large relative to the average amount of work it does between consecutive migrations.

Distributed Threads

Ariadne threads form the basic unit of computation in the distributed model. Each thread in the distributed environment is assigned a unique identifier during creation. Besides storing its own identifier, a thread also stores the identifier of the process in which it is created. When a thread dies – it either completes its function or is destroyed by another thread – the Ariadne system quietly notifies its creator process. Termination of a distributed computation is accomplished with the aid of this information. In all other respects, the distributed computing model is identical to the uniprocessor and shared-memory multiprocessor computing models.

Migration

Thread migration entails a higher level of architectural detail than that encountered in other aspects of thread support. Creating maps of run-time images of threads on heterogeneous machines is sufficiently complicated to render the effort not cost-effective, particularly with respect to portability. Despite this, process migration has received considerable attention in the research community. Proposals for its application include load sharing, resource sharing, communication overhead reduction, and failure robustness [27, 28]. Dynamic migration is usually addressed in the context of distributed operating systems, for example, V [29], DEMOS/MP [30].

At the current time, Ariadne supports thread migration on distributed environments of homogeneous machines. One proposal for thread migration simplifies addressing problems between machines by using a static preallocation of thread address spaces on all machines.

A newly created thread is assigned a slot within a large address space, and this assignment remains fixed for the life of the thread. Further, this slot must be reserved for the thread on all machines in the distributed system, just in case the thread makes an appearance at a machine. Such an approach is adopted by the Amber system [23]. A significant advantage of this approach is that migration of a thread from one machine to another does not require address translation of pointers on the stack. But this advantage is had at the expense of address space. It would not be possible for the system to host more than a given number of threads at a time. This is particularly unappealing in view of the fact that the limit is independent of the number of machines used. Additionally, data-parallel applications will suffer: space must be made available on all processors for threads that work on a domain located on a single processor.

The Ariadne system eliminates the restrictions described above by providing the user with a primitive for address translation of stack references by an immigrant thread. No stack space is pre-allocated for threads that migrate. When a thread migrates, its `tca` and essential stack are packed into a buffer and sent to the destination. Its shell is freed for reuse at the source. At the destination, the `tca` and essential stack are placed within a recycled shell, if one is available, or a new shell, if no free shells are available. Stack references are fairly easily identifiable as frame pointers and can be updated with an offset that is the difference between stack bottoms on source and destination machines. With this approach, Ariadne can host a large number of threads on a distributed system, a number that is directly proportional to the number of processors used and memory availability. Address translation occurs at low cost since the depth of function nesting in many applications – such as parallel simulation and other applications using Ariadne – is not high. The number of addresses to be translated is a linear multiple of the number of function nestings at the point of migration. Following address translation, a thread is placed on a queue of runnable threads on a destination process.

User defined references to objects placed on the stack (local objects) can be transformed in the same manner as frame pointers. The `a_updatep()` primitive is provided for this purpose. If compiler optimizations (e.g., common sub-expression elimination) are used, it is possible for some stack addresses to end up in registers. Because registers are not updated with the stack offset at destination processes, such optimizations are a threat to migration. One solution is to prohibit optimization for functions that can migrate. Another solution is to prohibit the use of (stack) addresses that could potentially end up in registers. Neither solution is completely satisfactory. But being practical, each solution is a small price to pay for distributed multithreading. A satisfactory solution would involve a post-migration update of register contents at destination processes.

Ariadne provides a simple interface to enable distributed computation, and in particular thread-migration, with any communications library. The thread-migration interface consists of two primitives: the `a_thread_pack()` primitive for use on a source process, and the `a_thread_unpack()` primitive for use on a destination process. The former primitive is used by the source to dissect and pack a thread into a buffer which is then sent over a network to the destination. To avoid race conditions that can occur when a thread attempts to make a copy of its own stack, a helper thread is asked to invoke the thread packing mechanism. By our assumption of processor homogeneity, data translation is unnecessary during message-passing.

The `a_thread_unpack()` primitive is used by a destination process to transform the data received into a runnable thread. When this thread is ready to resume execution for the first time on the destination process, context switching is accomplished via the `setjmp()/longjmp()` mechanism. Upon a return from a `longjmp()`, the thread is available for execution at the destination. A crucial difference here is that the `longjmp()` call executes on a host that is different from the host on which the `setjmp()` is called. Both

```
(01) void a_migrate(int procid)
   {
      int jump;    /* differentiates between source/destination */
      char *image;    /* thread image is placed here */
      int size;    /* size of the packed thread */
      thread_t self;    /* thread identifier */

(08) jump = a_thread_pack(ASELF, &image, &size);
      /* image now contains the thread tca and stack */
      if (!jump) {    /* on source if jump == 0 */
         a_set_migrate(ASELF);
         c_oubuf(OUTBUFID, BUF_SWITCH|BUF_TRUNCATE);
         c_pack(C_BYTE, image, size);    /* pack to form a message */
(14)     c_send(procid, THMIGR);    /* send the thread to procid */
         a_add_migrated();    /* update counters */
         free(image);    /* free image no longer needed */
         a_self(&self);    /* find identifier */
(18)     a_destroy(&self);    /* destroy thread on source */
      }
      else
         a_reset_migrate(ASELF);    /* on destination */

   }

   /* This function is called periodically by the Ariadne system.
   * Upon receiving a migrant thread, unpack it and convert it
   * into a runnable thread
   */
   void recv_messages(void)
   {
      char thread[THSIZE];
      char *thread_ptr;
      ...
(35)     if (c_probe(ANYPROC, THMIGR)) {    /* has a migrant arrived? */
         c_inbuf(INBUFID, BUF_SWITCH|BUF_TRUNCATE);    /* prepare buffer */
         c_recv(ANYPROC, THMIGR, BLOCK);    /* receive thread */
         c_unpack(C_BYTE, thread, c_rcvlen);    /* unpack message */
         thread_ptr = a_thread_unpack(thread, c_rcvlen);    /* create thread */
(40)     a_ready_thread(thread_ptr);    /* make it runnable */
      }
      ...
   }
```

Figure 10. Ariadne's thread migration with Conch support

calls are embedded in the `a_thread_pack()` primitive.

In Figure 10 is shown a simplified implementation of the `a_migrate()` function, using `a_thread_pack()`, `a_thread_unpack()`, and related support functions. In this example we utilize the Conch communications library [5] for message passing: functions with a `c_` prefix are Conch primitives. The thread is first placed into an "image" from where it is packed into a buffer that is sent across the network (see Figure 10, lines 8 –14). No longer required on the source, the shell of a thread that migrates is detached from the thread's function and recycled for later use within the `a_destroy()` call. Ariadne's D-processes poll input sockets for messages (line 35). When a message arrives, it is received in an input buffer, unpacked, and finally converted into a runnable thread (line 40).

Object Location and Migration

In Ariadne, objects are either encapsulated within a thread (local) or encapsulated within a process (global). Global objects are available to all threads hosted by a process. A thread typically works with its own local objects. Global objects are used to input initial values or store the results of computations. A thread is not permitted direct access to local objects belonging to other threads. To access another thread's local data a thread must synchronize or use a form of messaging based on global objects. If global objects are static, threads may locate global objects with the help of an object-locator mechanism (implemented on top of the Ariadne system, but available to the application). Such a mechanism may also be used when global objects are allowed to migrate, so that threads may ascertain their whereabouts when object access is required. A thread's local objects migrate along with the thread to enable it to continue execution unhindered at its destination.

In many applications an object-locator may be implemented in a fairly simple way. Consider, for example, a particle simulation on a grid: horizontal slices of the grid are viewed as global objects, and these objects are distributed across some number of distinct processes. An elementary calculation provides a mapping between particular grid locations and global objects. When a particle (thread) moving on a slice of the grid (global object) crosses a slice boundary, it requires access to the global object representing another slice of the grid. If the target object is resident on another process, the thread must migrate. After migration, it continues execution on the destination process until it needs to migrate again. How global objects are assigned to processes depends on the application and the user: the idea is to minimize communication. Good techniques for minimizing global state distribution is important – to enable large granularity in computations and infrequent migration between computations. The idea is to keep migration costs small relative to computation times.

Distributed Termination in Ariadne

A distributed computation terminates when there is no work left to be done by the distributed processes. If all Ariadne processes are idle and there are no messages in transit between processes, there is no work left to be done. Making this determination, however, is difficult. Though message buffers may be empty, messages may be in transit between processes; upon arrival, such messages may potentially reactivate Ariadne processes. The usual strategy is to base termination on a local predicate B_i which becomes true when every process P_i in the distributed system has terminated. Typical termination algorithms, such as described in [31], assume that processes are configured in special ways – rings, trees or predefined cycles, using topological information to aid in termination. A number of different algorithms have been proposed based on diffusing computations, ring structures, value-carrying tokens and time-stamping. A survey of such algorithms can be found in [32].

In Ariadne, each process determines its own status to be either *active* or *inactive*. An Ariadne process is *active* if:

- at least one of its own *offspring* (i.e., threads created within this process) is alive in the distributed system, or
- it currently hosts at least one offspring belonging to another Ariadne process.

If both conditions are false, an Ariadne process is *inactive*. Each process also maintains a count of the number of its own offspring that are alive (`live_count`) in the system, and an array (`destr_count`) – indexed by process identifier – that records the number of offspring of other processes destroyed at this process. On making a state transition from active to inactive, an Ariadne process uses the information in the `destr_count` array to send other processes its death-tolls of their offspring. Upon receipt of a message, each process reduces its `live_count` by this toll.

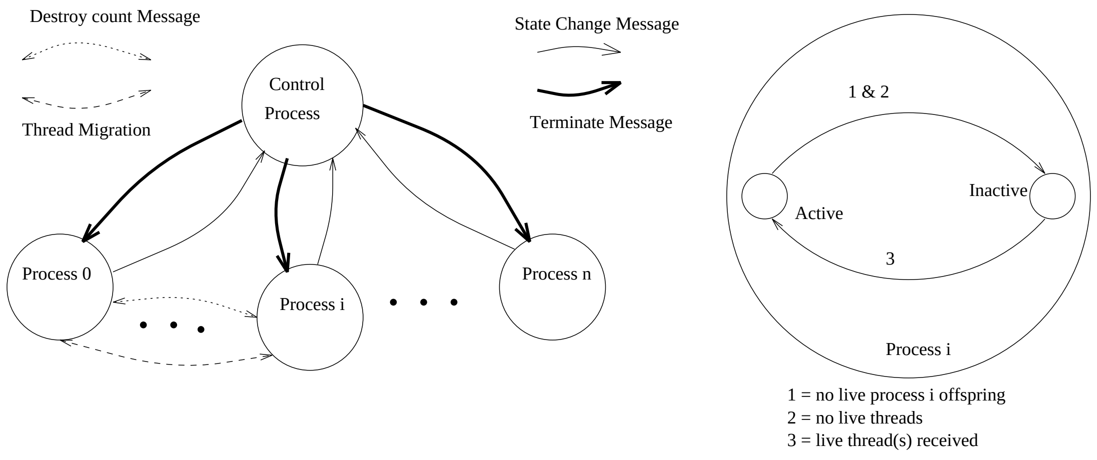

Figure 11. Termination in Ariadne

If a process with no threads finds its `live_count` at zero, it moves from the active to inactive state. State transitions between active and inactive states occur as shown in Figure 11. Whenever a state transition occurs, the transitioning process sends a message containing the label of the destination state to a termination controller process. The job of the controller is to track the number of active processes in the system. If all processes are inactive, the controller effects distributed termination: a termination message is sent to all Ariadne processes (see Figure 11). Upon receipt of such a message, each Ariadne process performs cleanup operations and terminates.

Distributed termination is initiated only when no user-created threads exist in the system. A message in transit containing a migrating thread is accounted for by the `live_count` variable at the thread's parent process. A message in transit containing other information is processed by appropriate user-created receptor threads at destination processes. These user-threads are also accounted for by the `live_count` variable.

Without assuming any process topology, Ariadne's termination algorithm effects termination when all processes are *inactive*. A key assumption of the distributed computing model is that computation must continue as long as user-created threads are alive on some process in the system. With reliable communication, this scheme has proven simple to implement, and effective. The controller process is chosen arbitrarily: with Conch we use a *front-end* (see [5]) process, and with PVM we use the first process that is created.

The termination algorithm is implemented using Ariadne's customization layer. If Ariadne processes frequently create and destroy threads, state transitions may be frequent. Messages enabling such transitions, and messages sent to the controller may result in high traffic. This may be avoided through the use of a decentralized termination algorithm, implementable within the customization layer. Within a process, Ariadne's support functions can be exploited to keep track of live thread counts and death counts of offspring of other processes. For applications in which thread populations remain fairly constant and threads undergo group destruction at the end of a run, the controller-based termination scheme works well.

PROGRAMMER INTERFACE

Ariadne's user-interface is simple. A user-program initializes the threads system with the `ariadne()` primitive. Once initialized, the invoker is transformed into a "main" Ariadne thread, and all Ariadne primitives become available to the application. After initialization, the user may create Ariadne threads and perform various thread-related tasks. The basic system primitives are shown in Table I. Details on the use of these and other parallel programming primitives, and illustrative examples, can be found in the Ariadne User Manual [33].

The main system thread may use `a_exit()` to signal termination. If no other user-created threads exist, the threads system terminates and the application may continue to run without Ariadne threads on a return from `a_exit()`. Otherwise, the main thread blocks until all other user-created threads terminate. Ariadne permits time-slicing among runnable threads at the same priority via the `a_defslice()` primitive. Primitives `a_limit()` and `a_buffer()` enable limits to be placed on the maximum number of simultaneously live threads, and number of thread shells on the free queue, respectively.

Threads are created with the `a_create()`, and destroyed with the `a_destroy()` primitives. A running thread is free to relinquish its CPU to another thread at any time: `a_yield()` is used if the target thread is a specific thread, and `a_yieldc()` is used if the target is an arbitrary thread at the same priority level. In the latter case, the scheduler is invoked to pass control between threads within the same priority class in FIFO order. The `a_setpri()` primitive enables a thread to change the priority of any live thread, including itself. As a result, the invoker may lower its priority and force a context-switch: control is given to the thread at the head of the highest-priority queue, or control remains with the thread whose priority was just changed. The `a_sleep()` primitive puts a thread to sleep in a delta queue for a given interval of time. A timer goes off when this time elapses, activating a handler which places the awakened thread in the ready queue. Several threads may use the delta queue simultaneously. Ariadne also provides primitives which enable an application to query threads. Primitive `a_ping()` returns the status of a thread: dead or alive, and `a_self()` returns a *thread_t* structure identifying the thread.

Counting semaphores have proven to be adequate in synchronization, for our current suite of applications. Mechanisms for spin-locks, mutexes and monitors are planned for a fu-

Table I. Basic Thread interface primitives

Initialization and Exit	
void ariadne(thread_t *thptr, int pri, int size);	void a_exit();
void a_buffer(int type, int size);	void a_limit(int size, int size);
void a_defslice(int type, int pri, int sec, int usec);	
Control	
void a_create(thread_t *thptr, void (*func)(), int ints, int floats, int doubles, ...);	
void a_yieldc();	void a_yield(thread_t th);
int a_setpri(thread_t th, int pri);	void a_destroy(thread_t th);
void a_sleep(int sec, int usec);	
Query	
int a_mypri();	int a_myid();
void a_self(thread_t *thptr);	int a_ping(thread_t th);
Synchronization	
sem* a_creates(int value);	void a_waits(sem *semptr);
void a_signals(sem *semptr);	void a_signalns(sem *semptr, int count);
void a_signalalls(sem *semptr);	int a_counts(sem *semptr);

ture release. The current implementation of semaphores enables semaphore creation (using `a_creats()`), waiting on a semaphore (using `a_waits()`) and signaling a semaphore (using `a_signals()`). Primitive `a_signalalls()` is used to signal all threads waiting on a semaphore, `a_signalns()` repeatedly signals a semaphore `n` times, and `a_counts()` returns a count of the number of threads currently blocked at a semaphore. The last primitive may also be used to determine if an `a_waits()` invocation will block. Similar primitives are available for Ariadne's shared-memory environment where semaphores allow threads running on different processors to synchronize.

Program execution

Upon invocation of the `ariadne()` initialization call, the main function is transformed into Ariadne's "main" thread with identifier 0. The priority of this thread is specified as a parameter in the call. The main thread continues to run until a thread with a higher priority is created or the function `a_exit()` is called. If a higher priority thread is created, the new thread runs after the main thread is placed on the ready queue. When the main thread invokes `a_exit()`, Ariadne checks for the existence of other user-threads in the system. If no other threads exist, the threads system terminates, and the application continues to run with a single thread of execution on returning from `a_exit()`. Otherwise, the main thread is placed in a special queue, awaiting termination of all other user threads. Primitive `a_exit()` must be invoked by the `main()` function or some function invoked by `main()`.

When Ariadne is initialized, it creates two internal threads: a *copier* thread and a *bridge* thread. The *copier* is used in saving essential stacks of threads that switch context on the common stack, and threads that migrate. The *bridge*, created with priority 0, allows the system to give control to an exiting main thread when all user threads are done. Because allowable user-thread priorities are positive integers, the *bridge* obtains control only when the main thread is blocked awaiting termination and no other user threads exist. On a uniprocessor, the *bridge* yields control to the main thread, causing it to exit the threads system. Single-threaded execution may continue from this point. On a shared-memory multiprocessor, a *bridge* must verify that no user threads exist on any Ariadne process before it can yield control to its main thread. This is done using the system's live-thread count, which is located in shared memory. This count is updated whenever a process creates or terminates a thread.

Table II. Basic Performance Results (Time in microseconds)

Operation	SPARCclassic		SPARCstation 20		Seq. Symm.	RS/6000	iPSC
	Sun-LWP	Ariadne	Sun-MT	Ariadne		Ariadne	
Null Thread	683	249	1130	40	363	90	466
Thread Create	249	57	1000	35	171	20	202
Thread Create (pre-allocated)	1199	16	-	11	204	10	19
Context Switch	58	66	12.5	15	91	14	16
setjmp/longjmp		27		3	16	5	2
Synchronization	-	158	55	20	121	25	45

PERFORMANCE RESULTS AND AN EXAMPLE

We measured Ariadne's performance on a number of machines: SPARC uni- and multiprocessors, IBM Risc workstations, Sequent Symmetry multiprocessors and Intel i860 (hypercube) processors. For lack of a well-accepted standard of comparison, performance is measured relative to the Sun lightweight process (Sun-LWP) library on SunOS 4.1, and the SunOS 5.3 multi threads (Sun-MT) library. We report the average time required to perform a number of basic operations over 1000 repetitions of each operation. The total time for all repetitions is computed using the `clock()` Unix library call.

The operations measured are the following. The *Null Thread* operation involves the creation and immediate execution of a null thread, with control returned to the creator. For this operation, stacks were preallocated. The *Thread Create* operation entails the creation of threads with and without preallocated stacks, respectively. In Sun-LWP, preallocation is done with the aid of `lwp_setstkcache()` and `lwp_newstk()` primitives. With Sun-MT, stacks are cached by default so that repetitive thread creation is cheap [34]. The Sun-MT threads library measurements are based on the default stack size allocated by the library.

Context Switch time is measured by timing context-switches between two specific threads. Timings for `setjmp` and `longjmp` operations are also reported, to give an indication of the additional work done by Ariadne during context-switching. *Synchronization* time is measured by forcing two threads to repeatedly synchronize with one another, as done in Reference [18]. Timings involving Ariadne were measured on the SPARCclassic (SunOS 4.1), SPARCstation 20 (SunOS 5.3), Sequent Symmetry, IBM RS/6000, and Intel i860 processors. Timings involving Sun-LWP were measured on the SPARCclassic, and timings involving Sun-MT were measured on a SPARCstation 20. All measurements are reported in Table II.

From the measurements, Ariadne appears competitive with Sun-LWP and Sun-MT threads on most operations. Sun's large creation time with stack allocation via `lwp_newstk()` may be attributed to setup of *red zone* stacks – protection against stack overflow [10]. The Sun-MT library also sets up protected stacks of default size 1MB. Ariadne's context-switching is a little more expensive than context-switching in Sun-LWP and Sun-MT, possibly because Ariadne invokes several functions during a context-switch. These include a reaper for processing dead threads, a delta queue scanner, an inspector which looks for stack overflow, a time-slice activator (if the time-slicing option is on), etc. Ariadne's context-switching overhead includes Unix's setjmp and longjmp overheads. There is scope for optimized context switching in Ariadne. For example, the deletion of function invocations and the use of assembler for special tasks will reduce Ariadne's context-switch overheads. All this is possible only at the expense of portability.

The cost of thread-migration includes the cost of packing a thread into a message with the help of the *copier* thread, destruction of the thread on the source, transmitting the message, receiving the message at the destination, and finally unpacking and converting the message into a runnable thread. We compute the average time taken to migrate a thread from one process to another and back, using two processes located on distinct machines on the same network. Results are averaged over six repetitions of an experiment, each involving 100 migrations, using two SPARCstation 5 and two Sequent Symmetry machines. Timings for (round-trip) thread migration, for threads with stacks of different sizes, are shown in Table III.

For purposes of comparison, Table III also includes the average time required for the round-trip transmission of messages (equal in size to the sizes of the thread stacks) between two processes. The base stack size used in each case is the size of a thread's essential stack. Average round-trip times were computed as described in the thread-migration experiment. Such a comparison reveals the extent of migration-related overheads over and above the cost of message-transmission. To determine the effects of larger stacks, the size of a dummy buffer – allocated locally by a thread – is increased. This forces the size of the thread's essential stack to increase.

Table III. Migration Time v/s Message-Transmission Time (milliseconds). The total elapsed time to migrate (send) a thread (message) from process A to process B and back from B to A is shown.

Machine	Function	Base Stack in bytes	Additional size in bytes			
			256	512	1024	2048
SPARCstation 5	Migrate	1176	11.31	11.11	13.98	16.07
	Transmit	1176	10.80	10.69	13.75	15.10
Sequent Symmetry	Migrate	684	77.03	83.11	95.73	114.76
	Transmit	684	74.41	80.46	93.91	111.23

From Table III it is evident that message-transmission time and related communication overheads dominate the migration activity. The relatively large migration and message-passing times on the Sequents reflect relatively high loads on the machines (i.e., 20–25 users were observed) when the experiments were conducted. These times were observed to increase by up to 45% with much higher loads (40 – 45 users). Overheads consumed by migration-related processing, prior to message-sending and after message receipt, account for roughly 1–3 % of total migration time. It is clear that faster processors and faster networks (e.g., ATM, Intel

Paragon) will yield smaller migration times.

Matrix multiplication on a multiprocessor

A simple example is used to illustrate the use of Ariadne on a shared-memory multiprocessor. This example involves matrix multiplication, and is meant to exhibit several features of Ariadne: shared memory management, scheduler customization, and support for a large number of threads. A similar example illustrating the use of Ariadne in a distributed environment can be found in Reference 33.

The program is given two matrices as input, with pointers `a` and `b` to the matrices declared on line 8 (see Figure 12). The program is to multiply the matrices and output a product. Each (i, j) entry in the product is computed by a distinct thread: it computes a dot product using row i of the first matrix and column j of the second matrix. This computation is performed using the function `dotProduct()` shown on line 39. With large matrices there is potential for a large number of threads to coexist in the system.

Matrices are stored in shared memory and are readily accessed by all Ariadne processes and their threads. Locking of shared data structures (matrices) is unnecessary because concurrent writes to a given memory location do not occur. The pointers to matrix elements are stored in a user created shared segment `usr_shmem_ptr` (line 20). Space for the matrices is allocated from the default shared memory segment provided by Ariadne (line 26). A total of `nprocs` Ariadne processes is created (line 17). Process 0 creates all the worker threads (line 30). Other Ariadne processes retrieve threads from the ready queue as they are created, and run them to completion. When process 0 has created all the required threads, it joins the other Ariadne processes in scheduling threads for execution. A thread terminates when it has produced some element(i, j) of the result matrix. The shells of terminating threads are returned to the free pool, available for reuse.

Because memory locking is not required, this application exhibits inherent parallelism. Further, because processors schedule threads from a common queue, balanced load enhances parallelism. Measurements indicate ideal speedups on a 4 processor SPARCstation 20 when the machine is dedicated to the application.

CONCLUSION

We have exploited multithreading using Ariadne on a number of hardware architectures with considerable success. In large measure, this success can be attributed to its portability. The layering scheme we chose to adopt has proven beneficial in a number of ways. The customization layer has enhanced our ongoing work in distributed simulation applications and enables further experimentation in threads-based distributed computing. We expect this layering to benefit a variety of applications that can exploit threads.

We believe Ariadne's thread migration mechanism to be unique. It enables a user to construct applications that can move threads between processes at will, enabling the implementation of strategies for load balancing and process-oriented computation. The exploitation of threads as active system objects tends to simplify parallel application development, as shown in the quicksort and matrix multiplication examples. Examples illustrating the use of Ariadne in shared- and distributed-memory environments can be found in Reference 8.

We have implemented and tested the Ariadne system on both uni- and multiprocessor environments, including the SPARC, Sequent Symmetry, IBM RS/6000, Silicon Graphics

```
 /* shmatmult.c - parallel matrix multiplication program */
#include "aria.h"
#include "shm.h"    /* shared segment allocator */
#include "mem.h"    /* default segment memory manager */
#include "schedshm.h"    /* custom scheduler */

#define SHM_KEY 100    /* user shared memory segment */
struct usr_shmem {double *r, *a, *b;};
struct usr_shmem* usr_shmem_ptr;    /* address of user shared segment */
int numRowsA, numColsA, numRowsB, numColsB, nprocs;
main()
{
   struct thread_t whoami;
   int i, j, i_id;

   a_set_shared();    /* use shared memory */
   a_shared_begin(nprocs);    /* begin nprocs number of processes */
   create_scheduler();    /* install a custom scheduler */
   /* get an additional shared segment of memory */
   usr_shmem_ptr = shmcreate(SHM_KEY, sizeof(struct usr_shmem),
         PERMS, &i_id);
   aria(&whoami, 7, 1024);    /* initialize Ariadne - priority 7 stack = 1024*/
   if (sh_process_id == 0) {
      /* read and store the matrices in the arrays a and b
      * matrices a, b, and r are allocated from primary shared memory */
      a = shmemalloc(sizeof(double)*numRowsA*numColsA);
      b = shmemalloc(sizeof(double)*numRowsB*numColsB);
      r = shmemalloc(sizeof(double)*numRowsA*numColsB);
      initialize_matrix();
      /* create a thread that computes each element of the matrix */
      for (i=0; i< numRowsA; i++)
         for (j=0; j< numColsB; j++)
            a_create(0, dotProduct, 5, SMALL, 2, 0, 0, i, j);
   }
   a_exit();    /* exit the threads system */
   print_results();
   /* free memory allocated for the matrices here */
   a_shared_exit();    /* stop children */
   shmfree(usr_shmem_ptr, i_id);    /* free user segment */
}

void dotProduct(int row, int col)
{
   double result=0.0;
   int i;

   for (i=0; i < numColsA; i++)
      result += *(usr_shmem_ptr->a+numColsA*row+i)*
   *(usr_shmem_ptr->b+numColsB*i+col);
   *(usr_shmem_ptr->r+numRowsA*row+col) = result;
}
```

Figure 12. Matrix Multiplication

workstations, and Intel i860 processors. Ports to the Intel Paragon environment are under consideration. Ongoing work seeks to exploit Ariadne for use with the *ParaSol* experimental (parallel) simulation system being developed at Purdue. Shared memory multiprocessing on Sequents has not been provided because of the unavailability of System V facilities on the Sequent. Nevertheless, Ariadne can be customized for parallel programming on the Sequent with help from the Sequent's parallel programming library.

REFERENCES

1. M. J. Rochkind, *Advanced Unix Programming*, Prentice Hall, 1985.
2. A. S. Birell, 'An introduction to programming with threads', *Technical Report 35*, DEC Systems Research Center, January 1989.
3. A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995.
4. V. S. Sunderam, 'PVM: a Framework for Parallel Distributed Computing', *Concurrency: Practice and Experience*, **2**(4), 315–339 (1990).
5. B. Topol, 'Conch: Second generation heterogeneous computing', *Technical report*, Department of Math and Computer Science, Emory University, 1992.
6. J. Sang, E. Mascarenhas, and V. Rego, 'Process mobility in distributed-memory simulation systems', G.W. Evans, M. Mollaghasemi, E.C. Russell, and W.E. Biles (eds.), *Proceeding of the 1993 Winter Simulation Conference*, 1993, pp. 722–730.
7. J. Banks and J. Carson, 'Process interaction simulation languages', *Simulation*, **44**(5), 225–235 (1985).
8. E. Mascarenhas and V. Rego, 'Migrant Threads on Process Farms: Parallel Programming with Ariadne', *Technical Report in preparation*, Computer Sciences Department, Purdue University, 1995.
9. D. Bertsekas, *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, 1989.
10. Sun Microsystems, Inc., Sun 825-1250-01, *SunOS Programming Utilities and Libries: Lightweight Processes*, March 1990.
11. D. Keppel, 'Tools and techniques for building fast portable threads packages', *Technical Report UWCSE 93-05-06*, University of Washington, 1993.
12. F. Mueller, 'A library implementation of POSIX threads under UNIX', *Proceedings of the 1993 USENIX Winter Conference*, January 1993, pp. 29–41.
13. S. Crane, 'The REX lightweight process library', *Technical report*, Imperial College of Science and Technology, London, England, 1993.
14. K. Schwan, et al., 'A C Thread Library for Multiprocessors', *Technical Report GIT-ICS-91/02*, Georgia Institute of Technology, 1991.
15. D. Stein and D. Shah, 'Implementing lightweight threads', *Proceedings of the 1992 USENIX Summer Conference*, 1992, pp. 1–9.
16. A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M Young, 'Mach threads and the unix kernel: The battle for control', *Proceedings of the 1987 USENIX Summer Conference*, 1987, pp. 185–197.
17. C. Thacker, L. Stewart, and E. Satterthwaite Jr., 'Firefly: A multiprocessor workstation', *IEEE Transactions on Computers*, **37**(8), 909–920 (1988).
18. M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, 'SunOS Multi-thread Architecture', *Proceedings of the 1991 USENIX Winter Conference*, 1991, pp. 65–79.
19. M. Vandevoorde and E. Roberts, 'Workcrews: An abstraction for controlling parallelism', *Int. J. Parallel Program.*, **17**(4), 347–366 (1988).
20. T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, 'Scheduler activations: Effective kernel support for the user-level management of parallelism', *ACM Transactions on Computer Systems*, **10**(1), 53–79 (1992).
21. B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, 'First class user-level threads', *Symposium on Operating System Principles*, 1991, pp. 110–121.
22. B. N. Bershad, E. D. Lazowska, and H. M. Levy, 'Presto: A system for object-oriented parallel programming', *Software–Practice and Experience*, **18**(8), 713–732 (1988).
23. J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield, 'The Amber System: Parallel programming on a network of multiprocessors', *Symposium on Operating System Principles*, 1989, pp. 147–158.
24. P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, and R. Chen, 'Distributed programming with

Objects and Threads in the Clouds System', *Technical Report GIT-GC 91/26,* Georgia Institute of Technology, 1991.

25. K. Chung, J. Sang, and V. Rego, '*Sol-es*: An object-oriented platform for event-scheduled simulations', J. Schoen (ed.), *Proceedings of The Summer Simulation Conference,* 1993.
26. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
27. C. Shub, 'Native code process-originated migration in a heterogeneous environment', *Proceedings of the ACM 18th Annual Computer Science Conference*, 1990, pp. 266–270.
28. J. M. Smith, 'A Survey of Process Migration Mechanisms', *ACM Operating System Review*, 28–40 (1988).
29. M. M. Theimer, K. A. Lantz, and D. R. Cheriton, 'Preemptable remote execution facilities for the V-system', *Poceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1985, pp. 2–12.
30. M. L. Powell and B. P. Miller, 'Process migration in DEMOS/MP', *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, October 1983, pp. 110–119.
31. E. Dijkstra and C. Scholten, 'Termination detection for Diffusing Computations', *Information Processing Letters*, **11**(1), 1–4 (1980).
32. M. Raynal, *Distributed Algorithms and Protocols*, John Wiley and Sons Ltd., 1988.
33. E. Mascarenhas, V. Rego, and V. Sunderam, 'Ariadne User Manual', *Technical Report 94-081*, Computer Sciences Department, Purdue University, 1994.
34. SunSoft, Inc., Sun 801-5294-10, *SunOS 5.3 Guide to Multithread Programming,* November 1993.