1995

Ariadne: Architecture of a Portable Threads system supporting Mobile Processes

Edward Mascarenhas

Vernon Rego
Purdue University, rego@cs.purdue.edu

Report Number:
95-017

ARIADNE: ARCHITECTURE OF A PORTABLE THREADS SYSTEM SUPPORTING MOBILE PROCESSES

Edward Mascarenhas
Vernon Rego

Computer Sciences Department
Purdue University
West Lafayette, IN 47907-1398

CSD-TR-95-017
March 1995

Ariadne: Architecture of a Portable Threads system supporting Mobile Processes *

Edward Mascarenhas
Vernon Rego
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Threads possess 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 computing on a variety of platforms. Ariadne is a portable system that exploits shared- and distributed-memory multiprocessors for parallel and distributed applications. Thread-migration is supported at the application level in homogeneous environments (e.g., networks of Sparcs and Sequents, Intel hypercubes). Threads may migrate between processes to access remote data, typically preserving locality of reference for computations with a dynamic data space. Ariadne provides a customization layer through which it can be tuned to specific applications: support for scheduling via a built-in scheduler or an application-specific scheduler, and a communication-library interface. Ariadne currently runs on the Sparc (SunOS 4.x and Solaris), Sequent, Intel i860, SGI, and IBM RS6000 environments. We compare the performance of Ariadne to the Sun lwp and Solaris thread libraries.

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

1 Introduction

A process is generally accepted to be an executing program with a single thread of control. Consider, for example, a typical Unix process: it proceeds along a single thread of execution, accessing data in its own address space. Interprocess or shared-memory communication may be used to provide data-sharing between two or more processes, allowing for either truly concurrent (on a multiprocessor) or pseudo-concurrent (on a uniprocessor) processing. The resulting computation may be viewed as multi-threaded, with two or more threads of control. Unfortunately, multi-threading via processes entails significant interprocess communication and synchronization effort and high context switching overheads.

In a multi-threaded computation, it is possible for a single thread to execute at a time (such as on a uniprocessor). Since a key requirement is for distinct threads to perform distinct activities asynchronously, a process may permit several threads of execution within itself: each thread executes as a mini-process, with a program counter and a stack for maintaining local variables and return addresses. In this view, distinct threads within a process have free and easy access to process variables, while ensuring privacy of their own variables. By keeping thread contexts small, thread control-switching cost can be made a fraction of process control-switching cost. A single process may host many threads within its address space, allowing for cheap and efficient multi-threading.

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 RS6000, SGI, and Intel iPSC environments. Support for parallel programming in shared memory multiprocessor environments, as well as 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. While current implementations support PVM [24] and Conch [28], other communication libraries may readily be used.

Ariadne was designed to support process-oriented parallel/distributed simulation [19]. Process-oriented simulation is popular because such models are known to map easily into implementations [3]. In this view, active objects in a real system are implemented as processes in a corresponding simulator. Being lightweight, threads are natural implementations of active simulation objects, affording inexpensive communication, switching and data sharing. Thus, threads can be used to implement models of active objects in simulations.

Parallel/distributed process-oriented simulation necessarily 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" (i.e., a packet in a data network) which executes on one processor and requires access to a passive object such as a "server" (i.e., a gateway in a network) whose simulation function is located on a remote host processor. We must

either move the required data to the requesting thread, or move the requesting thread to the data. In typical simulation applications, passive objects are sufficiently large (e.g., servers are often associated with queues, involving possibly complicated data structures containing transient active objects) to prohibit frequent data migration. On the other hand, threads are generally small, and thread-migration only entails moving thread state from one processor to another.

The design of Ariadne was motivated by three major goals. Being motivated by the need for process-oriented simulation support, a first goal was the support of thread-migration across homogeneous processors. Upon initiating a migration request, a thread executing on one processor resumes execution on a destination processor at the point where the request was issued on the source processor. Thread-state is preserved during the migration, and all objects local to the thread migrate along with the thread, being part of its constituent state. Efficient thread migration is crucial in a distributed computation to maximize performance gains, particularly when the granularity of computations between migrations are not large relative to the cost of migration. In the worst case, the cost of thread migration is shown to be equivalent to message-passing for simulation applications [19]. Thread migration simplifies the development of powerful parallel applications. Relocating computations so that they have access to remote data can be made transparent at the user-level, simplifying programming. Thread migration also provides a framework for experimentation with load-balancing strategies.

A second goal was to offer the benefits of portability and flexibility. Portability is important because of the cheap parallelism available via networked workstation (multi)processors. We exploit Unix library primitives (e.g., `setjmp()/longjmp()`) to avoid hardware dependencies in context-switching. Only a small part of thread initialization is coded in assembler. Ariadne is flexible in that, besides providing an efficient internal scheduler, it also allows for the use of a customized scheduler. Ariadne's internal scheduling policy is based on priority queues: at any given time, the highest-priority non-blocked thread runs. A variety of other policies may be obtained via such a customized scheduler. Ariadne provides several useful features including round-robin slicing among all runnable threads at the highest priority, thread "sleep" options, threads that run on common shared stacks and counting semaphores for synchronization.

A third goal was to provide a facility for multi-threaded distributed computing [13]. By providing a clean interface between Ariadne and communication primitives, a variety of distributed computing tools may be used in conjunction with Ariadne. Distributed initialization and termination, thread migration and message passing are supported at a layer above the basic Ariadne kernel. The system has been used for diverse parallel applications, sometimes 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 a particle-physics application a thread may represent one or more particles; in SOR applications [5] a thread may represent computation at one or more grid points; in simulations of personal communication systems a thread may represent a mobile phone. The functions of dynamic system objects are naturally representable by mobile threads: computations which can migrate between processes. The above applications are not exhaustive, and Ariadne may be used in a variety of parallel and distributed applications.

1.1 Related Work

Threads can be supported at the user level or within the kernel. Each approach has its advantages and disadvantages. Some examples of user space threads systems are Sun-lwp(SunOS) [23], POSIX threads [15], and Quick Threads [11]. User-level threads do not require kernel intervention and can therefore be managed cheaply. User-level threads also have the advantage of being relatively simple to implement, are portable and flexible; they can be modified without making changes to the OS kernel. Unfortunately user-space threads can perform poorly in the presence of I/O and page faults: when a thread blocks, its host process also must block until the the I/O is done, a page is loaded, or the requisite operating system (OS) service has been granted.

Kernel-space threads do not cause a host process to block when a service is required of the OS. When a thread blocks, the kernel may switch control to a runnable thread, if one exists. Because kernel-threads can impair performance if 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 Solaris [22], Mach [25], and Topaz [26] 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 on top of kernel-threads. Systems exemplifying this approach include Solaris(Sun-MT [16]), Mach(CThreads), and Topaz(WorkCrews [29]). User-threads are multiplexed over available kernel-threads in a multiprocessor environment.

An idea advocated recently is the combination of kernel-space and user-space threads, to obtain the advantages of both. Scheduler activations [2], and Psyche [12] are examples of such systems. In scheduler-activations, kernel-threads do up-calls into user space to give 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. Observe that parts of the operating system kernel must be modified if these mechanisms are to be implemented.

In Ariadne, threads are used as the basis for parallel and distributed computation. On shared-memory multiprocessors, Ariadne multiplexes user threads on top of multiple processes. Another system that works in this way is PRESTO [4]. Ariadne threads access remote objects by migrating to the location of the object. Experiments reported in [19] indicate that this scheme provides good performance. Advantages of this approach include one-time transmission and increased locality of reference. Programming with threads however requires more care and discipline than ordinary programming [6]. For example, if a threads system does not enforce protection between thread stacks (one thread's stack may overflow onto another thread's stack, if the first has not been given enough space) resulting bugs can be hard to locate.

Other systems providing parallel programming with threads are Amber [7] and Clouds [9]. As in Ariadne, these systems provide threads and objects as basic building blocks. Ariadne migrates local objects that belong to the address space of a thread (allocated on a thread's private stack) along with

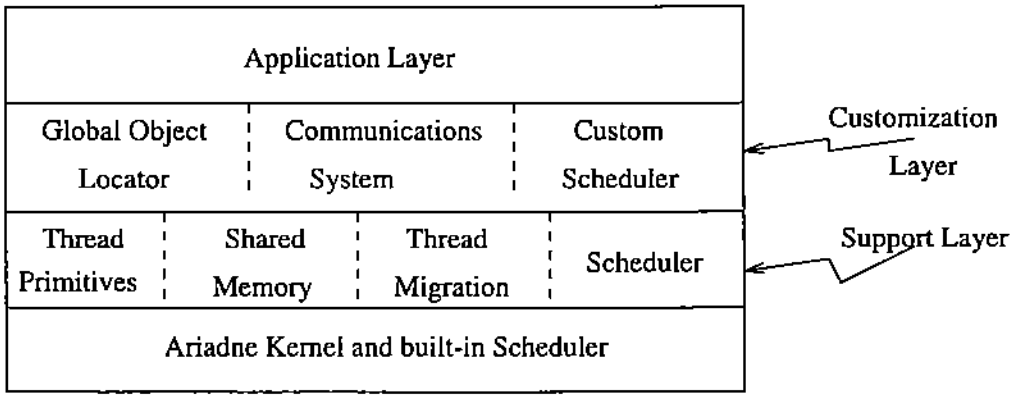

Figure 1: Ariadne Architecture

the thread during migration. Frequent access of local objects motivates this methodology. Locations of shared global objects (outside the thread space) are handled at a layer above Ariadne. This allows Ariadne to be used with software systems that provide efficient means for mapping and locating objects in a distributed environment. By providing a neat interface between the threads library and a parallel programming environment, with the requisite support for multi-threading, Ariadne allows the creation of tailored solutions.

2 Overview of Ariadne

The Ariadne system consists of three layers: the lower-most layer contains the kernel, the middle layer provides thread-support, and the top layer provides customization support (see Figure 1). 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. use of Ariadne for simulation or real-time applications. The Sol [8] simulation library uses Ariadne for scheduling and handling events – the next thread to run handles 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 a priority level 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 support shared memory, thread migration, distributed multi-threaded computations, and specialized schedulers. When using customized schedulers, Ariadne's kernel calls user functions (up-calls) to get the next runnable thread; if a running thread blocks or suspends execution in 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 mapping software may be used. For distributed computing, PVM, Conch or equivalent communication software may be used.

2.1 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 Unix `setjmp()/longjmp()` primitives. 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 on the number of threads used at a given time, allowing 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 simple function call. Transparent thread migration and transparent scheduling of threads on available processors makes for a simplified parallel and distributed 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-mapping software.

- **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 with Unix primitives. But performance measures indicate the difference is not significant, and more than offset by the resultant gain in portability. For efficiency, Ariadne implements context-switching via a function, instead of a scheduler thread. Though this introduces race conditions and complicates the implementation, such complexity is transparent to the user.

2.2 A Simple Example

We illustrate the use of Ariadne with a simple example on sorting, i.e., the well-known quicksort [1]. The sequential algorithm is initiated from the main function with the call QUICKSORT(S).

```
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 size n integers using quicksort, a pivot is chosen to divide the array into two parts – the first part containing elements with value less than the pivot, and the second part containing elements with value greater than the pivot. The quicksort routine is then called recursively on the two parts. An Ariadne program which implements this on a shared-memory multiprocessor is shown in Figure 2. The conversion is natural: routine quicksort in the sequential program simply becomes a thread in Ariadne. The main function sets up the shared environment using `a_set_shared()` and `a_shared_begin()`. Function `create_scheduler()` implements a shared memory based scheduler which allows multiple processes to obtain runnable threads from a common queue. The `ariadne()` call initializes the threads system.

The main function (at priority 7) creates one `quicksort()` thread and exits (actually suspends its own execution until no more threads are left) via `a_exit()`. The `a_create()` primitive is used to create new threads. At this point, the `quicksort()` thread (at priority 6) runs, partitions the array into two and creates another thread to sort elements smaller than the pivot. The parent thread continues with a recursive call, to sort elements larger than the pivot. When all threads but main have completed their work and exited the system, the main thread returns from `a_exit()` and the sorted array may be printed.

This program will run on any supported architecture without modifications. Load balancing is automatic – a single queue stores work left to be done in the form of waiting threads. Each process obtains a runnable thread from the head of the queue. Threads migrate transparently between processes; in this example they are created by one process and may run inside another. The example does not use Ariadne's synchronization mechanisms: each thread works on a distinct piece of the array.

2.3 Basic Kernel Mechanisms

A Unix process may host a large number of threads. Information corresponding to each thread is stored in a thread "shell" consisting of a thread context area (`tca`) similar in appearance to a process control block (which the OS maintains for each process) and an associated stack space. Within a process, each thread is identified by a unique identifier. This identifier is also unique to a distributed system of processes. Depending on user specification, each thread either runs on its own stack or runs on a *common stack* that

```
#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 */

     /* 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);
     }
     a_exit();
     /* print results */
     a_shared_exit();
     shmfree(aa, shm_id);
}
```

Figure 2: Quicksort Program in Ariadne

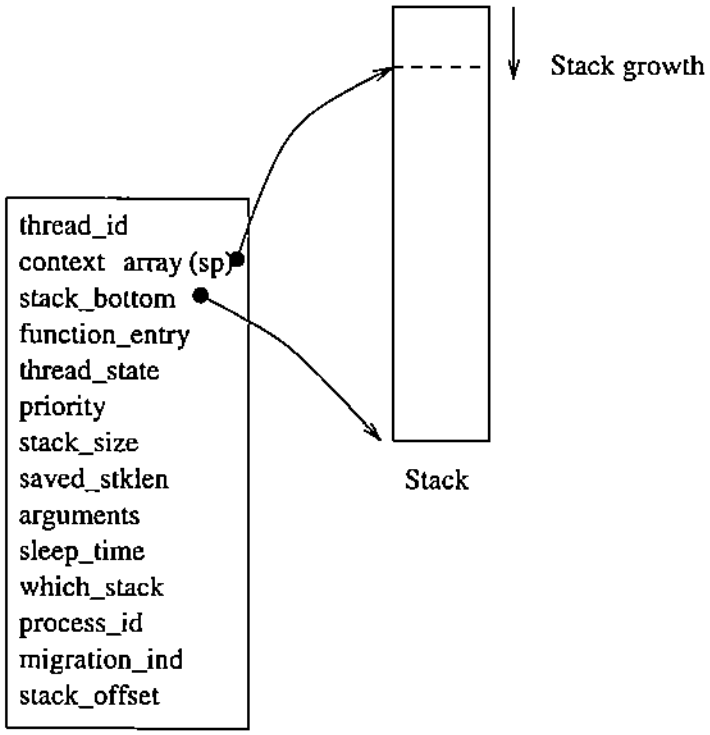

Figure 3: Thread Context Area (`tca`)

is shared by all threads. The `tca` contains a pointer to this stack, and to a context array where registers are saved. The size of this array is determined by the size of `setjmp`'s buffer (declared in `setjmp.h`). Thread state is saved at and restored from this area for context-switching. The stack pointer (`SP`) is also stored in this array. Other values stored in the `tca` include the function entry (corresponding to the thread), argument list, thread priority, thread state, and stack-related values (see Figure 3).

Thread Creation and Destruction

A thread is created from a thread shell by associating a function with the shell, and specifying the size of the stack on which the thread is to run. The `a_create()` primitive performs the binding and places the thread on a ready queue of runnable threads at the specified priority level. A thread may be destroyed when it no longer needs to run, or when it has run to completion. Its shell (`tca` and stack) is placed on a free list for possible reuse by another thread with similar stack size requirements. Ariadne allows the creator of a thread to pass a thread a variable number of integer, float, and double type arguments. Arguments are stored in the `tca` until the thread obtains control for the first time.

```
/*
 * 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 4: Context Switch

Stack Allocation

Stack sizes are flexible. These may be passed as an encoded size (a multiple of a fixed size) or as a user-specified value. For efficiency, a lazy stack allocation scheme is adopted: stack space is allocated only when a thread is scheduled to run. Thus threads which are created but never run are prevented from consuming stacks. Nevertheless, memory can be a serious problem when several runnable threads compete for large independent stacks. Such a situation is circumvented in Ariadne through the use of a *common stack*, allocated in static memory. All threads, or threads with large dynamic memory requirements can run on this stack. On a context-switch, a thread's essential stack (i.e., portion of stack in use) is saved in a private area, typically much smaller than the size of stack requested by the thread. As a result, overall memory requirements are reduced. An additional cost is incurred here: saving and restoring a stack on each context-switch whenever a thread runs on a common stack.

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 specified fraction of the total stack size.

Context-Switching Mechanism

Context-switching is performed on the stack of a thread that relinquishes control (a "yielding" thread). A segment of code demonstrating this idea is shown in Figure 4. Specific Ariadne primitives which effect

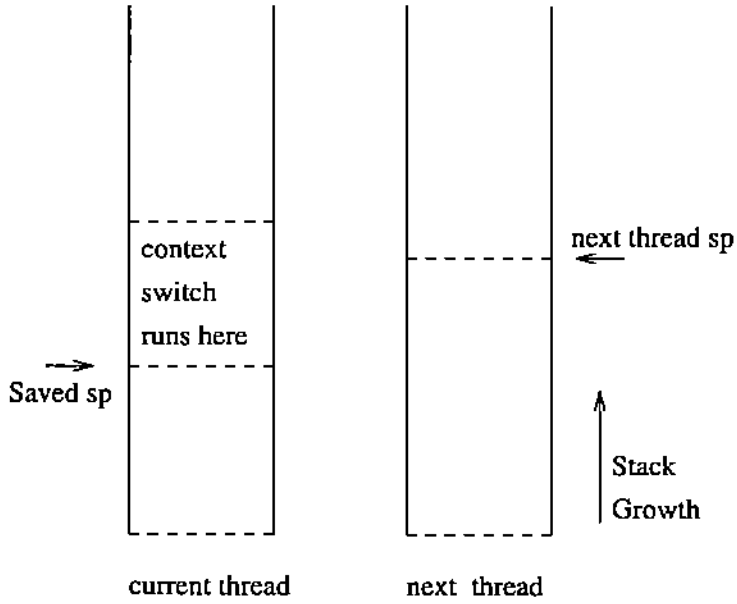

Figure 5: State of stacks during a context switch

context-switching include functions `a_resched()` and `a_yield()`. First the thread relinquishing control is placed on the tail-end of its priority queue. It is marked *READY* only after the switch is complete. Because the context-switch is performed on the yielding thread's stack, marking it *READY* before the switch is complete can create race conditions on a multiprocessor. When the switch is complete, the yielding thread is marked *READY* by the target thread (i.e., thread selected to run next). In Figure 5 is shown the state of the old and new threads' stacks during a context-switch.

As shown in Figure 4, a `setjmp()` causes the yielding thread's context, `SP` (stack pointer) and `PC` (program counter) to be saved in the `tca`, with a return value of 0. An ensuing `longjmp()`, with a given `tca` as parameter, causes the corresponding thread to resume execution at the point where it invoked `setjmp()`, returning a value of 1. The return value determines whether a context-save or a context-restore has occurred, deciding the course of action to be taken next. On some architectures (e.g., IBM RS6000), `setjmp()/longjmp()` primitives prohibit jumps from lower to higher addresses (or vice versa, if the direction of stack growth is reversed), necessitating some hand-coding. Observe 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 thread migrate primitives.

2.4 Scheduling in Ariadne

The Ariadne system provides a built-in scheduler that allocates portions of a host process's time-slice to its distinct threads. An Ariadne thread operates at one of several integer priorities. Within a priority class scheduling is FIFO. At any given time, the highest priority runnable thread runs. 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 and displaying the value of a function, and another thread for reading from a socket. 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 – execution time or elapsed time based. Slice granularity can be set using the `a_defslice()` primitive.

Critical Sections Handling in Ariadne

Interrupts which occur during thread execution can affect control flow. Providing protection for Ariadne's internal data structures is crucial. When a thread completes its execution slice, or when a sleeping thread awakens, the interrupt handler gets control. In such cases Ariadne uses a flag to protect critical sections that manipulate important data structures. If the flag is found set when an interrupt occurs, the interrupt handler returns control to the thread without handling the interrupt. Such handing may be resumed when safe. Being interrupted in a critical section allows a thread to run for an additional time-slice, so that it may exit the critical section. Masking interrupts before entering a critical section is possible, but involves an expensive Unix system call.

Ariadne provides users with `a_interruptoff()` and `a_interrupton()` primitives, based on masking mechanisms, for protection of critical sections in applications. 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. If time-slicing is being used, the scheduler obtains control from the thread when the critical section is exited. This allows for an equitable distribution of the execution slices among runnable threads. These primitives are also useful in preventing interrupts from occurring during non re-entrant C library calls, e.g. `printf()`.

Scheduler Customization

Ariadne was designed primarily for parallel and distributed simulation applications. Based on event occurrences in time, threads are required to be scheduled for execution in increasing order of time-stamps. Each thread is viewed as a handler for a sequence of events, and runs when its next scheduled event occurs. Because Ariadne's internal scheduler does not provide for time-based and other scheduling, an interface is provided for the development of customized schedulers. As a simple example, consider a thread which attempts to compute $n!$ by creating another thread to compute $(n-1)!$ and then returns n multiplied by the value computed by the child thread. Such a computation requires threads to be scheduled for execution in LIFO order.

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

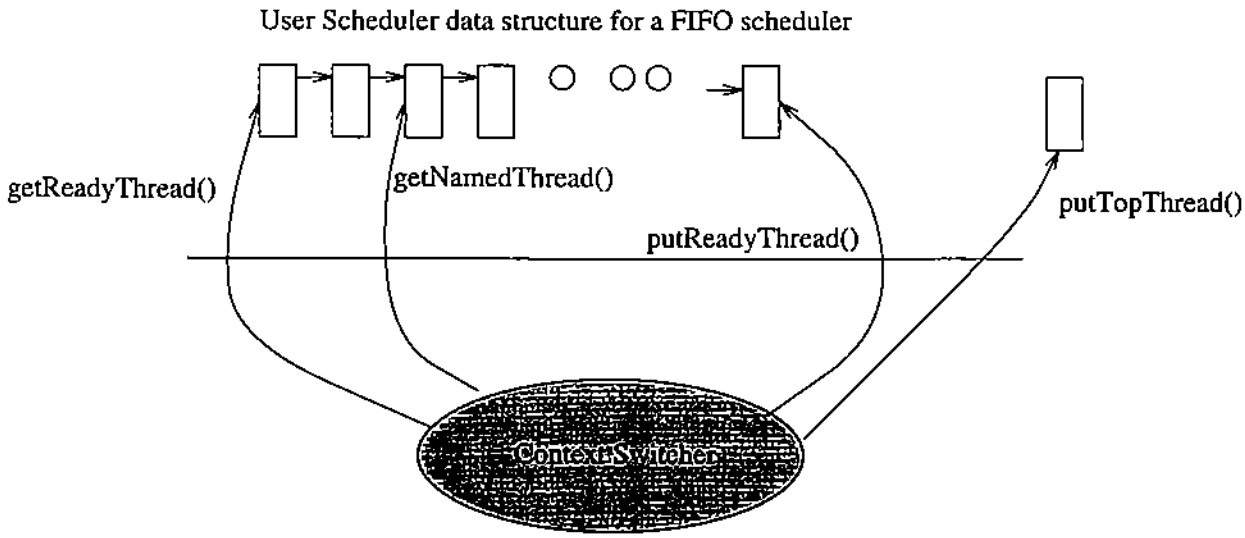

Figure 6: Scheduler Customization

example, the schematic for a FIFO scheduler is shown in Figure 6. The user must provide four functions callable by the Ariadne kernel during a context switch, and also maintain `tcas` 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 gives control to the thread. The `putReadyThread()` primitive returns a thread which has just run to the user. This function must insert the thread into an appropriate position in the data structure, so that it may be subsequently retrieved for possible execution when `getReadyThread()` is invoked. What these functions do will depend on the scheduling policy required by the application. In Figure 6, `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 at this point or allow the Ariadne kernel to reclaim it on its free lists. The remaining two primitives may be used in special situations. If the Ariadne kernel needs to temporarily suspend execution of a thread while some internal function (e.g., check-pointing of thread state) is being 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. For this the data structure used must support removal of a specific thread based on a key.

3 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

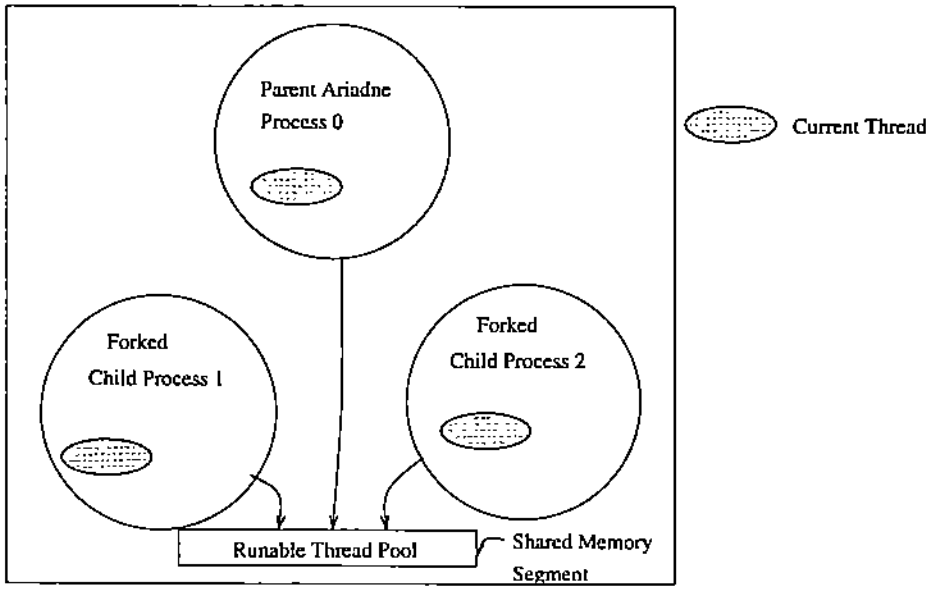

Figure 7: Ariadne Multi-processor Configuration

exploits multiprocessing power by multiplexing threads on distinct processes, generally using as many processes as there are available processors. Besides high creation costs during setup, a disadvantage of using processes instead of kernel-supported threads is that a thread blocking on a system call makes its host process also block, thus blocking other threads[1]. The Ariadne system is initialized via an initialization primitive in a "main" Ariadne process. On shared-memory multiprocessors, this main process forks off a user-specified number of processes: each is assigned an integer identifier `sh_process_id` and is granted access to shared memory segments. The parent process is assigned the identifier zero (see Figure 7), and the child processes are assigned identifiers 1, 2, 3 etc. These identifiers permit the 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 operate on shared-memory. Each process maintains its own stack and notion of "main" thread which is not sharable among processes. At any given time, each process may run a distinct active thread, independently of the other processes. Shared-memory synchronization constraints may cause dependence 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 highest-priority 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 create/free additional shared memory segments with the

[1]Ariadne is currently being interfaced with kernel-level support, where available, to provide for non-blocking execution without sacrificing portability.

`shmcreate()/shmfree()` primitives. Dynamic shared memory areas can be obtained from the primary segment for user data via `shmemalloc()/shmemfree()`. These are based on inter process communication 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 Section 6.

Thread Scheduling

The multiple Ariadne processes execute as user processes on a multiprocessor. The OS schedules each of these on arbitrary processors as they become available. An unavoidable problem at the user-level[2] arises in the following situation: the OS schedules a process with a blocked thread (one that is waiting for another thread to signal a semaphore). Each such schedule causes a cycle to be wasted, until the process hosting the thread holding the resource continues execution. At this point, the resource may be released and the process hosting the blocked thread will 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. This requires that the shared structure be locked, accessed and then unlocked, to prevent multiple and simultaneous accesses by different processes. The scheduler customization feature described earlier may be used to implement such a 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. Here it is possible for two threads at differing priority levels to be running at the same time. This scheduler is used in an example given later (Section 6.1). In the other, the scheduler always accesses the highest priority queue. Only when no threads at this priority exist will a thread waiting in a lower priority queue be scheduled. In this case it is possible for an Ariadne process to idle, even though ready threads at lower priority levels are available. A process will not run a thread whose priority is lower than the priority of a thread that is being run on another process.

Race Conditions

Race conditions are inherent to thread executions on shared-memory multiprocessors due to the differing rates at which processes make progress. Protection against race conditions is necessary for execution sequences that are inconsistent. For example, thread `A` must be protected from *yielding* control to thread `B`, if the latter is already running on some processor, or is currently blocked on some request. As another example, thread `A` should be protected from destroying thread `B` if the latter is currently blocked or executing on some processor.

Ariadne ensures that race conditions do not arise during a context-switch from one thread to another. There is a finite amount of time between the instant at which execution of one thread is suspended and

[2]Ongoing work interfacing kernel-level support with Ariadne seeks to avoid this problem.

the instant at which another thread begins to run, on a given processor. During such a time, both the thread being suspended and the thread scheduled to run next are said to be in states of "transition". Once a thread is placed in a transition state, all internal Ariadne operations on the thread are forced to completion without interruption. For example, consider a typical 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 on other processors. 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.

4 Distributed-Memory Multiprocessor and Network Support

Ariadne may be used for multi-threaded computations on distributed memory multiprocessors: networks of uniprocessors and/or shared-memory multiprocessors, and distributed memory multiprocessors. We enable distributed multi-threading by enhancing the basic architecture of Ariadne with additional layered functionality. The distributed environment may be configured according to user needs, perhaps utilizing a user's favorite communication library or distributed computing software system. A clean and simple interface allows Ariadne to be used flexibly with arbitrary communication software systems. Prototype implementations of distributed-Ariadne include support for PVM 3.3.4 [24] and Conch [28].

On a distributed-memory system Ariadne processes are identified with integers starting at 0. Shared-memory support may also be exploited if a shared-memory multiprocessor is available on the distributed system. If Ariadne processes fork off children on a uni- or multiprocessor, the parent process is responsible for handling all communication with other hosts in the system. Child processes are always identified using an integer-tuple `<process identifier, shared process identifier>`. A graphical depiction of this architecture is shown in Figure 8. Ariadne (parent) processes communicate via messages, using primitives from the underlying communications library (PVM, Conch etc). Child processes always interact with parent Ariadne processes via shared-memory. This results in significant cost savings, since shared-memory based inter process communication is cheaper than socket-based communication. Messages bound for other hosts are placed in a shared message queue. Parent processes retrieve messages from this message queue and send them off to destination hosts. Parent processes may either be dedicated to communication or share computational load with children. The choice will depend on the message density of a given application.

4.0.1 The Distributed Programming Model

Ariadne's distributed model consists of three components: Ariadne processes, Ariadne threads and user objects. Objects may be simple, such as integers. Examples of complex objects include data-structures,

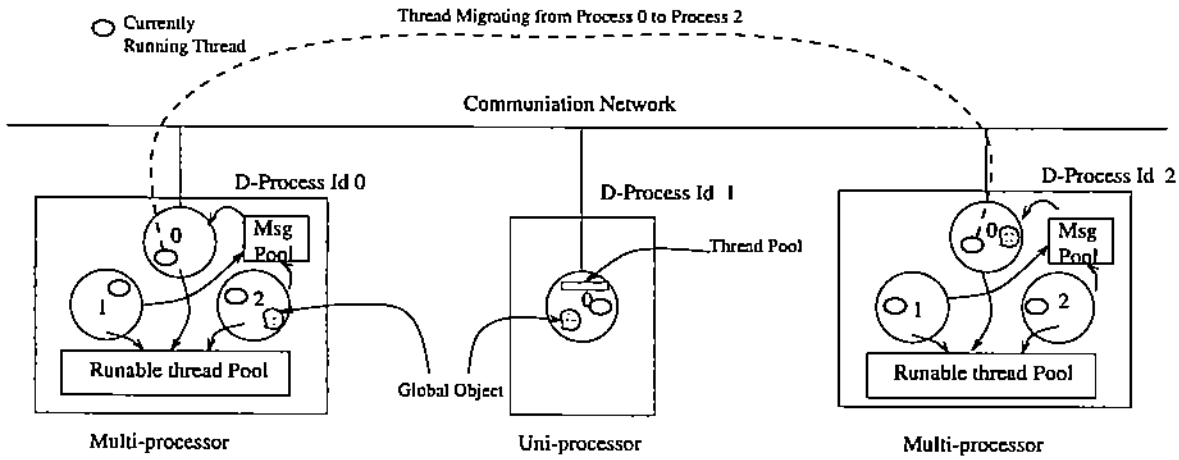

Figure 8: Distributed Ariadne

such as lists or trees. Ariadne processes encapsulate Ariadne threads and global objects: threads are active, mobile computational units, with freedom to move between all Ariadne processes in the distributed environment. Typically, threads move to access (global) objects located on other Ariadne processes (i.e. computations that chase data). Objects resident within an Ariadne process are considered global to the threads within the process. Such objects are static, and there is generally no motivation to migrate such objects. For thread-migration, Ariadne depends on the use of an object locator mechanism: a migrating thread needs to know which host it must migrate to in order to access required data.

An Ariadne thread encapsulates objects that are local to it; all local objects migrate along with their parent thread when a thread migrates. In earlier work [19] we have shown that thread migration offers several advantages in applications such as parallel simulation. 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 and round-trip transmission time, which proves to be costly [19]. If remote objects are to be repeatedly accessed by a thread, it is preferable to migrate the thread to the process hosting the object.

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. A thread stores its own identifier and also the process identifier of its creator in its `tca`. The Ariadne system uses this information to notify a process when a thread created by the process terminates, or is destroyed within another process. The distributed computation relies on a termination algorithm based on this information. In all other respects distributed-Ariadne threads are identical to Ariadne threads on uni- and shared-memory multiprocessor systems.

Thread Migration

In general, thread migration involves 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 from the research community. Proposals for its application include load sharing, resource sharing, communication overhead reduction and failure robustness, among others [20, 21]. Dynamic migration is usually addressed in the context of distributed operating systems, for example V [27], DEMOS/MP [17].

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 [7]. 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 memory. 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 because space must be made available on all processors for threads that execute on each domain.

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; a new shell is created at the destination (either requiring a stack from a free queue of stacks, or a new stack if the free queue is empty). 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. User defined references of objects placed on the stack (local objects) can also be transformed in this manner. Ariadne provides users with the `a_updatep()` primitive to perform this function. As a result of 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 well-structured applications 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 the queue of runnable threads on a destination process.

Ariadne provides a simple interface to allow for distributed computation, in particular thread-migration, with any communication library. The thread-migration interface consists of two primitives: a `a_thread_pack()` primitive to be used on the source process, and a `a_thread_unpack()` primi-

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

8          jump = a_thread_pack(ASELF, &image, &size);
           /* image now contains the thread tca and stack */
           if (!jump) {            /* we are 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)) {              /* is there a migrant ? */
                   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 9: Ariadne thread migration with Conch support

tive on the destination process. The former primitive is used by the source process to dissect and pack a thread into a buffer which is then sent over a network to the destination. Because processors are assumed to be homogeneous, data translation involving byte ordering is unnecessary. The latter primitive is used by a destination process to transform the data received into a runnable thread. When this thread eventually resumes execution, but for the first time on the destination process, context switching is accomplished via the `setjmp()/longjmp()` mechanism. 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 calls are embedded in the `a_thread_pack()` primitive. Upon a return from a `longjmp()`, the thread is available for execution at the destination. To avoid race conditions that can occur when a thread attempts to make a copy of its own stack, it is necessary for the system to request a helper thread invoke the thread packing mechanism.

In Figure 9 is shown a simplified implementation of the `a_migrate()` using primitives `a_thread_pack()`, `a_thread_unpack()` and related support functions. In this example we utilize the Conch communications library 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 9, lines 8 –14). No longer required on the source, the shell of the thread that migrated is detached from the thread's function and stored for reuse via the `a_destroy()` primitive. Conch processes poll input sockets for messages (see line 35). When available, a message is received in an input buffer, unpacked and finally converted into a runnable thread (see line 40).

Object Location and Migration

In Ariadne, objects are either local to a thread (encapsulated within a thread) or global (encapsulated within a process, but available to all threads hosted by the process). A thread typically works with its own local objects; global objects are used for initial input values or to update results of computations. Direct access of local objects belonging to other threads is not permitted. This may be done by synchronization or a form of messaging using global objects. Global objects are generally static, located by threads via an object locator mechanism (implemented on top of the Ariadne system, but available to the application). Such a mechanism must also be used when global objects are allowed to migrate, so that threads may ascertain their current location 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, the object locator is 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 requires to migrate again. How global objects are

assigned to processes depends on the application and the user: the idea is to minimize communication. Minimization of global state distribution to minimize migration and enlarging computation granularity are important: the idea is to keep migration costs small relative to computation times.

Distributed Termination in Ariadne

In general, effecting termination of a general distributed computation involves global information that is nontrivial to obtain – a runtime determination of inactive Ariadne processes and a simultaneous absence of messages in transit between processes. Though message buffers may be empty, messages still in transit may potentially reactivate Ariadne processes. The usual strategy is to base termination on a local predicate B_i which in turn becomes true when each process P_i in the distributed system has terminated. Typical termination algorithms (see, for example, [10]) assume that processes are configured in special ways, such as rings, trees or predefined cycles, using topological information to aid in termination. A number of different algorithms have been proposed based on diffusing computation, ring structures, value-carrying tokens and time-stamping. A survey of such algorithms can be found in [18].

In Ariadne, each process ascertains whether its own status is *active* or *inactive*. An Ariadne process considers itself *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.

An Ariadne process considers itself *inactive* when both the conditions listed above are not satisfied. Each process also maintains a count of the number of its live offspring (`live_count`) in the system, and an array (`destr_count`) indexed by process identifier, that stores the number of offspring of other processes destroyed at this process. When an Ariadne process makes a transition from an active to inactive state, it scans the `destr_count` array to send every other Ariadne process a count of the number of that process's offspring it has destroyed. Each process reduces its `live_count` by this number, upon receiving such a message. If a process that is not host to any threads finds its `live_count` reaching zero, it makes a transition to the inactive state. Transitions between active and inactive states take place as as shown in Figure 10. Whenever a process makes a transition from one state to the other, a message containing the label of the destination state is sent to a termination controller process whose job is to tracks the count of active processes. When the controller finds all processes inactive, it effects distributed termination by sending a termination message to all Ariadne processes (see Figure 10). Upon receiving such a message, Ariadne processes perform cleanup operations and terminate.

Observe that distributed termination is guaranteed to take place *only* when there are no more live user-created threads in the system. Messages in transit representing (migrating) threads are accounted for via the `live_count` variable, monitored by the thread's parent Ariadne process. Messages in transit

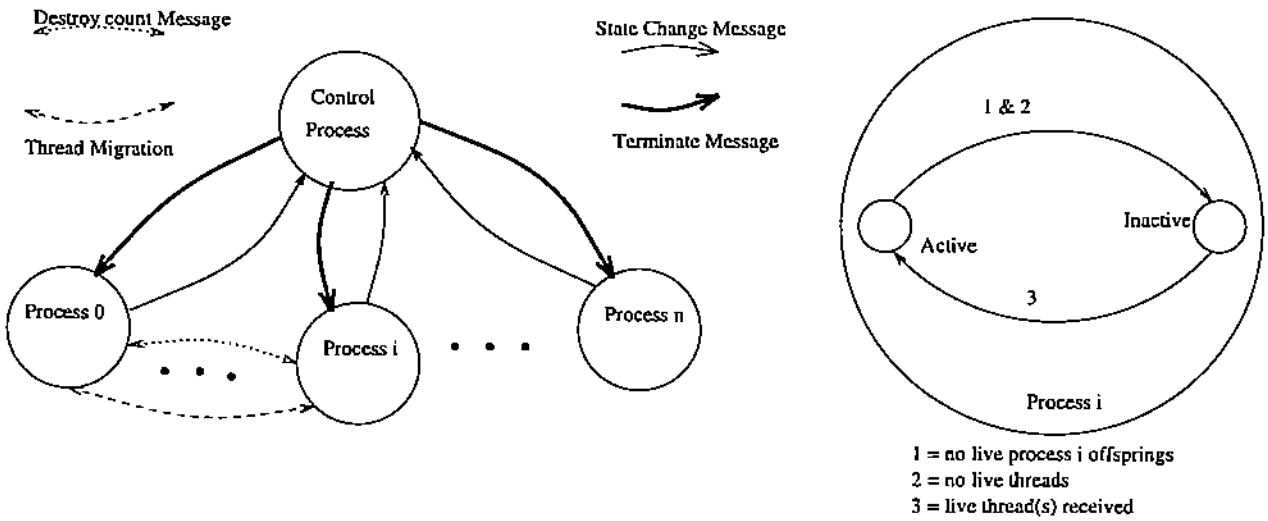

Figure 10: Termination in Ariadne

representing information other than threads must be processed by appropriate user-created receptor threads at destination processes; being user-threads, these are also accounted for by the `live_count` variable.

Without a topological requirement on processes, Ariadne's termination algorithm effects termination when all Ariadne processes are deemed *inactive*. A key assumption made in the distributed computing model is that computation must continue as long as user-created threads are still alive and resident on some process in the system. With a reliable communication network, this scheme has proved to be simple to implement and effective. The controller process which effects termination is chosen arbitrarily: in Conch we use a *front-end* (see [28]) process, and in PVM we use the first process that is created.

The termination algorithm described above is implemented using the the customization layer of Ariadne. It is possible that Ariadne processes may create and destroy threads with some frequency, causing processes to make many state transitions between active and inactive states. Accounting information enabling such transitions, and information sent to the controller process may result in high message traffic. To circumvent this, a decentralized termination algorithm may be used, also implementable within the customization layer. If thread population remains fairly constant during a run, and threads tend to be destroyed in a group towards the end of a run, the termination scheme we employ works efficiently. Accounting information is sent only when processes change state and/or when threads are destroyed. Ariadne's support layer provides functionality that can be exploited at the customization layer to keep track of live thread counts in a process and the number of offspring of other threads destroyed by a process.

5 Programmer Interface

The user-interface to the Ariadne system is simple. A user-program initializes the thread system through the use of the `ariadne()` primitive. Once initialized, the invoking application is transformed

Initialization and Exit	
void aria(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);	sem* a_creates_sh(int value);
void a_signals(sem *semptr);	void a_signalns(sem *semptr, int count);
void a_waits(sem *semptr);	int a_counts(sem *semptr);
void a_signalalls(sem *semptr);	

Table 1: Basic Thread interface functions

into a "main" Ariadne thread, and all Ariadne primitives are available to the application. After system initialization, the user typically creates Ariadne threads and uses Ariadne primitives to perform various thread-related tasks. The basic system primitives are shown in Table 1. Details on the use of these primitives, along with examples, can be found in the Ariadne User Manual [14].

Ariadne permits round-robin time slicing among runnable threads at the same priority via the `a_defslice()` primitive. Use of `a_exit()` guarantees thread-system termination, provided all user-created threads have either terminated or been destroyed. The application continues to run without Ariadne threads from this point. Otherwise, the main thread remains blocked until no user-created threads exist, at which time it returns from `a_exit()` and the application continues without Ariadne threads. 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.

Thread control primitives must be invoked for manipulating control between threads. Threads are created with the `a_create()`, and destroyed with the `a_destroy()` primitives. A running thread is free to relinquish a CPU to another thread at any point in its execution: `a_yield()` is used if the target thread is a specific thread, and `a_yieldc()` is used if the target thread is any other thread at the same priority level. In the latter case, control is passed from thread to thread within a priority class in first-come first-served order, and necessarily entails invocation of the scheduler. The `a_setpri()` primitive enables a thread to change the priority of any live thread, including itself. As a result, the invoker may

drop in priority and force a context-switch: control is given to the thread at the head of the highest-priority queue, or control is given to the thread whose priority was just changed. The `a_sleep()` primitive puts a thread to sleep for a given interval of time in a delta queue. A timer goes off when the interval elapses, activating a handler which places the awakened thread in the ready queue. Several threads may be asleep simultaneously in the delta queue.

Ariadne provides a number of primitives that enable an application to make thread-related queries. Primitive `a_ping()` returns the status of a thread: dead or alive. Similarly, `a_self()` returns to the invoker a *thread_t* structure which identifies the thread.

A set of simple thread synchronization primitives are provided. Counting semaphores have been found to be adequate for our current suite of applications. We intend to provide mechanisms for spin-locks, mutexes and monitors in a future release, as dictated by application needs. 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()`). These actions are equivalent to the **P** and **V** synchronization primitives available on some operating systems. 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. Here the semaphores allow synchronization between threads running in different processes.

Program execution

As indicated earlier, upon invocation of the `ariadne()` initialization primitive, the main function is transformed into Ariadne's "main" thread, with thread identifier zero. The priority of this thread must be specified as a parameter in the `ariadne()` call. This thread continues to run until a thread with a larger priority is created or a call to `a_exit()` is made. If the main thread creates a thread with a higher priority than itself, the main thread is placed on the ready queue and henceforth treated just like any other thread. If the main thread invokes `a_exit()`, Ariadne checks for the existence of other-user threads in the system. If no other live user-threads exist, the thread-system is terminated, and the application continues normally with a single thread of execution upon a return from `a_exit()`. Otherwise, the main thread is placed in a special termination queue, awaiting the termination of all other user-level threads. Primitive `a_exit()` must be invoked by the `main()` function or some function invoked by `main()`.

The system creates two internal threads: a *copier* thread and a *bridge* thread. The *copier* is used to save stack contents of a thread which relinquishes control while running on the common stack, or to save stack contents of a thread that requires migration. The *bridge* is created during system initialization and is assigned a priority of zero. Since the only allowable user-level priorities are integers greater than zero, the *bridge* thread obtains control only when the main thread is blocked in the termination queue and no other user-level threads exist. In the uniprocessor case, the *bridge* yields control to the main

Operation	Sparc-Classic		Sparc 10		Seq Symm	RS6000	iPSC
	Sun-lwp	Ariadne	Sun-MT	Ariadne	Ariadne		
Null Thread	683	249	1130	40	363	90	466
Thread Create	249	57	1060	30	171	20	202
Thread Create (pre-allocated)	1199	16	1800 (Bound)	10	204	10	19
Context Switch	58	66	12.5	19	91	14	16
setjmp/longjmp		27		3	16	5	2
Synchronization	-	158	55	20	121	25	45

Table 2: Basic Performance Results (Time in microseconds)

thread, causing it to exit the Ariadne system. The user may continue to perform thread-free computation from this point. On a shared-memory multiprocessor, a running *bridge* must ensure that no user-level threads exist on any Ariadne process before it can yield control to its main thread. This is accomplished by placing the system's live-thread count variable in shared memory; an update of ths count is done whenever some process creates or destroys a thread, or a thread terminates execution. Termination of Ariadne on a distributed memory system was explained in the previous section.

6 Performance Results and an Example

We report on the performance of Ariadne ports to a number of machines: Sparc uni- and multiprocessors, IBM Risc workstations, Sequent Symmetry multiprocessors and Intel i860 (hypercube) processors. For lack of a standard of comparison, we measure our performance relative to the Sun lightweight process library (Sun-lwp), available on SunOS 4.1.3C, and the Solaris threads (SUN-MT) library. We measure the average time required to perform a number of basic operations over 1000 repetitions of each operation. Total time for all repetitions is measured using the `clock()` Unix library call.

The operations of interest are described as follows. The *Null Thread* operation involves the creation and immediate execution of a null thread, causing a transfer of control back to the creator. For measurement of this operation, stacks were preallocated on all environments. The *Thread Create* operation entails the creation of threads without and with preallocated stacks, respectively. In Sun-lwp, preallocation is done with the aid of `lwp_setstkcache()` and `lwp_newstk()` primitives. Measurement of *Context Switch* operations are based on context-switching actions between two specific threads. Measurement of setjmp and longjmp operations are provided to give an indication of Ariadne's context-switching overhead outside of Unix overhead. The time for the *Synchronization* synchronization operation is measured via synchronization between two specific threads which repeatedly synchronize with each other, as done in [16]. All measurements are reported in Table 2. Measurements involving Ariadne were made on the SPARC classic (SunOS4.1), SPARC 10 (Solaris), Sequent Symmetry, IBM RS6000, and Intel i860 processors. Measurements involving Sun-lwp were made on the SPARC classic (SunOS4.1) and the Sparc

Machine	Function	Base Stack in bytes	Additional size in bytes			
			256	512	1024	2048
SPARC Classic	Migrate	1176	12.5	12.5	14.1	15.2
	Transmit	1176	12.3	12.4	14.0	15.8
Sequent Symmetry	Migrate	684	31.4	31.5	34.8	47.8
	Transmit	684	29.7	31.8	32.0	47.8

Table 3: Performance of Migration (Time in milliseconds)

10 (Sun-MT, Solaris multi-threaded system).

The measurements indicate that Ariadne competes well with Sun-lwp and Solaris threads on most operations. Sun's large creation time with stack allocation via `lwp_newstk()` may be attributed to setup of *red zone* protected stacks – checks for stack overflow [23]. Ariadne's context-switching time is slightly greater than context-switching time in Sun-lwp and Solaris. Ariadne invokes several functions during a context-switch, including a reaper function for processing dead threads, a function that looks for awakened processes on the sleep (delta) queue, a function that checks for stack overflow, a function for time-slice activation (if the time-slicing option is used) etc. Included in Ariadne's context-switching overhead is the overhead of Unix's setjmp and longjmp mechanism. Because our emphasis has been on portability and not optimization, there is considerable scope for optimization of context switching overhead in Ariadne. For example, deletion of function invocation related to the many checks during a context switch reduces Ariadne's context-switching time to match Sun-lwp's. Coding some functions at assembly level can be expected to reduce this time further.

In Table 3 we report the time (in millisecs) required to perform thread migration. To establish a base for comparison, we also report the time required to transmit a message of the same size as the thread. Such a comparison will reveal the extent of migration related overheads exclusive of message-sending. The average time taken by two processes to migrate a thread between one another over a total of 1000 times was measured. To account for a variation in network traffic, this experiment was repeated 6 times.

The thread-migration time includes the time required for packing a thread into a message, message-transmission, message-receipt, and finally thread unpacking and conversion into runnable form. The same experiment was performed to measure the average time required to transmit messages the same size as the message representing the thread (this time, of course, excluding processing related to thread migration). The base stack size in each case is the size of a thread's essential stack. To examine the effect of larger stacks, we simply increase the size of a dummy buffer allocated locally by a thread, causing the size of its essential stack to increase.

From Table 3 it is evident that message-transmission time and related communication overheads dominate the migration activity. Overheads consumed by migration-related functions, prior to message-sending and after message receipt, account for roughly 1% of total time. It is clear that faster processors

and faster networks (e.g., ATM, Intel Paragon) will yield smaller migration times.

6.1 Matrix multiplication using a multiprocessor

In this section we provide a simple example of an application utilizing Ariadne on a shared-memory multiprocessor. The application involves matrix multiplication, and is meant to illustrate several features of Ariadne: shared memory management, scheduler customization, support for numerous threads. A similar example illustrating the use of Ariadne in a distributed environment can be found [14].

The program is given two matrices as input, with pointers a and b to these matrices declared on line 8 (see Figure 11). The program is to multiply the matrices and output the product. Each (i, j) entry in the product is computed by a distinct thread: its task is to compute a dot product using row i and column j of each of these two input matrices, respectively. This thread performs the computation using the `dotProduct()` function shown on line 38. Thus there is potential for a large number of threads to co-exist in the system.

Matrices are stored in shared memory and are readily accessed by all Ariadne processes, and consequently by threads they host. 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 completes creation of the requisite number of threads, it continues scheduling threads for execution, as is done by the other Ariadne processes. The termination of a thread corresponds to the production of some (i, j) element in the result matrix. Threads that terminate are returned as shells to the free pool, available for subsequent creation.

Because memory locking is not required, this application exhibits inherent parallelism. Further, balancing load across processors by allowing processes to independently schedule threads for execution enhances parallelism. Measurements on this simple application indicate linear speedup on a 4 processor Sparcstation when the machine is dedicated to the application.

7 Conclusion

We have exploited multi-threading with the Ariadne system 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 as well as support for multi-threading in distributed computing software experimentation. We expect this layering to benefit a variety of applications that can utilize threads.

```
(1) /* 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 */
(10) 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 */
(20)     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++)
(30)                    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)
    {
(40)     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;
(47) }
```

Figure 11: Matrix Multiplication

The proposed thread migration mechanism is currently unique to the Ariadne system. 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. Details on the use of Ariadne in shared- and distributed-memory environments are reported in a companion paper [13].

We have tested the Ariadne system on both uni- and multiprocessor environments, including Sparc, Sequent, IBM RS6000, SGI 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, based on the Sequent's parallel programming library.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] 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, February 1992.

[3] J. Banks and J. Carson. Process interaction simulation languages. *Simulation*, 44(5):225–235, May 1985.

[4] 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, August 1988.

[5] D. Bertsekas. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.

[6] A. S. Birell. An introduction to programming with threads. Technical Report 35, DEC Systems Research Center, January 1989.

[7] 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. In *Symposium on Operating System Principles*, pages 147–158, 1989.

[8] K. Chung, J. Sang, and V. Rego. *Sol-es*: An object-oriented platform for event-scheduled simulations. In *Proceedings of The Summer Simulation Conference*, 1993.

[9] 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.

[10] E. Dijkstra and C. Scholten. Termination detection for Diffusing Computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[11] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[12] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First class user-level threads. In *Symposium on Operating System Principles*, pages 110–121, 1991.

[13] E. Mascarenhas and V. Rego. Parallel Programming with mobile threads in Ariadne. Technical Report in preparation, Computer Sciences Department, Purdue University, 1995.

[14] E. Mascarenhas, V. Rego, and V. Sunderam. Ariadne User Manual. Technical Report 94-081, Computer Sciences Department, Purdue University, 1994.

[15] F. Mueller. A library implementation of POSIX threads under UNIX. Winter USENIX, pages 29–41, January 1993.

[16] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *Proceedings of the 1991 USENIX Winter Conference*, pages 65–79. Sun Microsystems Inc., 1991.

[17] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. Proceedings of the Ninth ACM Symposium on Operating Systems Principles, pages 110–119, October 1983.

[18] M. Raynal. *Distributed Algorithms and Protocols*. John Wiley and Sons Ltd., 1988.

[19] J. Sang, E. Mascarenhas, and V. Rego. Process mobility in distributed-memory simulation systems. In *Proceeding of the 1993 Winter Simulation Conference*, pages 722–730, 1993.

[20] C. Shub. Native code process-originated migration in a heterogeneous environment. In *Proceedings of the ACM 18th Annual Computer Science Conference*, pages 266–270, 1990.

[21] J. M. Smith. A Survey of Process Migration Mechanisms. *ACM Operating System Review*, pages 28–40, July 1988.

[22] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of the 1992 USENIX Summer Conference*, pages 1–9. SunSoft, Inc., 1992.

[23] Sun Microsystems, Inc., Sun 825-1250-01. *SunOS Programming Utilities and Libries: Lightweight Processes*, March 1990.

[24] V. S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.

[25] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach threads and the unix kernel: The battle for control. In *Proceedings of the 1987 USENIX Summer Conference*, pages 185–197, 1987.

[26] C. Thacker, L. Stewart, and E. Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[27] 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, pages 2–12, 1985.

[28] B. Topol. Conch: Second generation heterogeneous computing. Technical report, Department of Math and Computer Science, Emory University, 1992.

[29] M. Vandevoorde and E. Roberts. Workcrews: An abstraction for controlling parallelism. *Int. J. Parallel Program.*, 17(4):347–366, August 1988.