1995

Migrant Threads on Process Farms: Parallel Programming with Ariadne*

Edward Mascarenhas

Vernon Rego

Purdue University, rego@cs.purdue.edu

Report Number:

95-081

Mascarenhas, Edward and Rego, Vernon, "Migrant Threads on Process Farms: Parallel Programming with Ariadne*" (1995). *Department of Computer Science Technical Reports*. Paper 1253.
<https://docs.lib.purdue.edu/cstech/1253>

**MIGRANT THREADS ON PROCESS FARMS:
PARALLEL PROGRAMMING WITH ARIADNE***

**Edward Mascarenhas
Vernon Rego**

**Department of Computer Sciences
Purdue University
West Lafayette, IN 47907**

**CSD TR-95-081
December 1995**

Migrant Threads on Process Farms: Parallel Programming with Ariadne *

Edward Mascarenhas
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

We present a novel and portable threads-based system for the development of concurrent applications on shared and distributed memory environments. Implementing user-space threads, the Ariadne system is highly effective for medium to coarse grained applications. Sequential programs are readily converted into parallel programs for shared or distributed memory, with low development effort. We describe basic threads primitives, and constructs for synchronization and computation in concurrent applications. Ariadne flexibly caters to a variety of communication environments, through a simple interface. It supports the development of customized schedulers on shared memory multiprocessors, and offers a thread migration capability for distributed environments. Scheduling of computations at the thread level offers both task- and data-driven executions. Thread migration is a powerful feature which turns remote memory accesses into local accesses, enables load-balancing and simplifies program development. Ariadne currently runs on the SPARC (SunOS 4.x, SunOS 5.x), Sequent Symmetry, Intel Paragon, Silicon Graphics IRIX, and IBM RS/6000 environments. We present Ariadne's parallel programming capability through several examples, reporting on performance measurements obtained with each.

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

1 Introduction

As distributed computing technology continues to evolve, providing for future computing environments, developers of distributed-support software face serious challenges. Future environments will consist of a heterogeneous mix: multiprocessor desktops, specialized workstations and fast, multiprocessor servers. These will communicate over high-speed networks such as the 100-Mbps Ethernet, FDDI or ATM. Concurrency supporting environments [42] such as PVM [37] and P4/Parmacs [7] must also evolve, to exploit newer technologies and provide novel services. Though concurrent computing on loosely coupled machines has repeatedly proven to be both low-cost and effective [29, 37], metrics of performance and parallelization effort exhibit both a need and scope for improvement. Execution performance of a distributed application invariably hinges on network latency and throughput. Computation speeds are orders of magnitude larger than communication speeds, and typical network environments are relatively slow. Good performance is achieved only when machines communicate and synchronize infrequently, with fewer and larger messages interspersed with computations of large granularity.

Given definite hardware constraints (e.g., bandwidth, machine speed), there is a serious need for improved performance through good software design. Besides obvious enhancements like the overlap of communication and computation, dynamic task-scheduling, balanced processor loads and reduced synchrony, there are other considerations. These include the handling of unpredictable, and outside (user) influences on system loads, and network and Operating System (OS) effects. For example, employing the generic functionality of an OS for a message-passing process- or task-based computation, as done in PVM and P4, poses problems of efficiency. Kernel-based process scheduling for layer invocation, combined with client insensitivity to critical timing in protocol functions, can effect significant though avoidable overheads [10, 43].

Typical distributed applications, such as those supported by PVM and P4, are typically limited to process- or subroutine-based computation. Proposals for their redesign/extension, to include threads support, have begun to appear [14]. While processes provide flexibility, their management and scheduling overheads tend to dominate potential gains, and application-specific distributed management is cumbersome. For simplicity, distributed applications often consist of communicating processes which compute using a subroutine structure. But though subroutine calls are inexpensive, they lack flexibility in that they run only when invoked, and when invoked they run to completion.

We propose the use of cheap, user-space threads [38, 25] as the unit of computation in shared- and distributed-memory environments. The chief advantage of a threads system lies in its ability to deliver mixed-mode execution: a process may alternate between sequential and parallel execution phases, in particular turning to one or more eligible threads when a running thread blocks on a system call. On a uniprocessor, threads offer interleaved but sequential execution sequences, with relief from blocking system calls. On a shared memory multiprocessor, threads also offer parallel execution sequences. All threads within a process communicate using global data, and thus keep synchronization costs low. Threads in distinct processes communicate using shared memory or some form of interprocess communication (IPC). Clusters of shared memory multiprocessors with a few CPUs can create and execute tasks cheaply, improve asynchrony, facilitate load distribution, and overlap communication

with computation.

A distributed process may run as a multithreaded process. Threads become basic units of computation and scheduling. A thread executes using its own program counter and stack for storing local variables and return addresses, with free access to all process variables. When stack requirements are unpredictable, threads may run on large, shared stacks. When thread contexts are small, a process may host thousands of threads. The development of a threads-based application is guaranteed to cost less than its multi-process counterpart. Task schedules impossible to obtain with subroutine structures now become possible. Finally, because of flexibility in task schedules and simplicity of thread operations, programming with threads holds promise for reduced parallelization effort.

We propose a paradigm of domain-oriented computation, based on threads and events, to enable low-cost parallelization and rapid experimentation. Indeed, this motivated development of the Ariadne threads system [25] to support 'process-oriented' parallel simulation [22]. The idea is to move domain functionality into a layer distinct from parallel simulation functionality. We found threads functionality highly useful in supporting multiple domains simultaneously. By encapsulating difficult domain functionality, users are relieved of the burden of recreating domain code. Experiments describing efficiency gains with such layering in protocol design can be found in [16].

Threads enable a direct and natural expression of concurrency in many applications. In particle-physics, for example, a thread models a molecule; in personal communication systems, a thread may represent a mobile phone; in queuing or teletraffic simulations, a thread may represent a request. Threads typically model active and mobile objects. Program development is often simplified, because emphasis may now be placed on how basic computational units (threads) interact. Ideally, considerations of thread scheduling, load balancing, and remote data access are handled by the threads system.

Distributed threads are bound to generate requests for remote data access. A thread on one host will ask for data on another host. We may satisfy the thread in one of two ways: either move the data to requester's host, or move the requesting thread to the data. Passive data-objects are often large, making frequent data migration costly. In data parallel computations, for example, it is usual practice to divide a domain among processors; in teletraffic simulations, switches may host multiple queues. In each case, an object's data may be large and sufficiently complicated by internal transient active objects to discourage frequent data migration. On the other hand, threads are generally small [15], and thread migration is cheaper than data migration when locality of data access is an issue, or when thread state is small compared with data state [18]. In our experience with parallel simulation applications, the cost of moving threads between processors is roughly the same as the cost of messaging the same event-related information [33].

The Ariadne system supports threads-based programming on shared- and distributed-memory environments. The system consists of a base library and a set of interface modules for applications and communication subsystems. The base library provides a range of constructs to alleviate programming effort: primitives to create and destroy threads, to set concurrency levels, to synchronize between threads, to transfer control between threads and to migrate threads. Modules are available for the support of shared schedulers, and Ariadne currently provides an interface supporting the PVM [37] and Conch [41] communication libraries. The system is written in C, and supports applications developed in Fortran, C and C++.

Figure 1: Ariadne and ParaSol architectures

A brief overview of the uniprocessor system is provided in Section 2. Details on the system's internal mechanisms for thread migration and distributed support can be found in [25]. Shared- and distributed-memory models are presented in Sections 3 and 4, respectively. The customization interface, using PVM as an example, is described in Section 5. Four examples exemplifying Ariadne's use are detailed in Section 6, along with discussions on performance. Section 7 discusses related work, and a brief summary and future plans are given in Section 8.

2 The Ariadne System: An Overview

The Ariadne system consists of three layers: the bottom layer contains the kernel, the middle layer provides threads support, and the top layer provides customization support. This layering, along with layered support of the ParaSol parallel simulation system [22], is shown in Figure 1. The **kernel layer** facilitates thread creation, initialization, destruction, and context-switching; it uses an internal priority-queue based scheduler. The **support layer** enables applications to use threads via supervised kernel access and allows customization. The system can be tailored to support shared memory, thread migration, distributed computations and specialized schedulers. The **customization layer** provides independent modules that aid in customization, such as the scheduler customization used in our parallel simulation work. Support-layer software is an intrinsic part of the threads library, while the customization modules are meant to flexibly cater to application needs.

The layered design enables the system to interface well with application-level software. The ParaSol system [22] is one example of this, and the Clam active-messaging system [16] is another. Consider, for example, threads support in ParaSol. Custom modules in ParaSol are mapped into Ariadne's kernel and support layers, as shown in Figure 1. The global object locator is an Object Manager that maps unique object identifiers onto logical process identifiers. These are then mapped onto processor identifiers in the communications subsystem. At present, the latter is either the PVM or Conch library, though others may also be used. The custom scheduler is simply ParaSol's *simulation driver*. It schedules threads by delivering to Ariadne a thread which handles the lowest time-stamped event in the system. The layering has been very useful in ParaSol's development.

2.1 Basic Mechanisms

A Unix process supports threads in one of two ways: in user-space or in kernel-space. Ariadne is an example of a user-space threads system, while Solaris [36] is an example of an OS that provides kernel-space threads. With kernel support, an OS may schedule threads. If one kernel thread blocks, due to I/O, page faults, etc., an unblocked kernel thread can run. With user-space threads, however, a blocking thread forces its host (process) to block, preventing the host from running other eligible threads.¹ Because nonblocking execution requires OS intervention through kernel traps, kernel threads are expensive. In contrast, user threads are both cheap and highly efficient. Finally, because not every OS provides kernel threads, applications based on kernel-space threads are not portable. Scheduler Activations [2] and Psyche [21] are examples of systems that attempt to combine the advantages of both user-space and kernel-space threads, through use of up-calls and software interrupts.

Ariadne keeps thread information inside a thread “shell”. This shell includes a thread context area (*tca*), similar in appearance to the process control block an OS maintains for each process, and an associated stack. Within a process, each thread is identified by a unique identifier, also unique to a distributed system of processes. A thread runs on its own stack, allocated either from shared memory or from the process heap. It may also run on a large *common stack* shared with other threads, where threads with large stack requirements run. For this, the active portion of a thread’s stack must be saved before a context switch, and copied into the common stack before thread execution is resumed. Besides having access to static or global data and data on the stack, heap, and shared memory, each thread has a *private* data area of a fixed size. This area may be used to store useful object attributes. Data in the private area is accessible from within any nested function invoked by the thread, but is not accessible to other threads.

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 binds the function to the thread and places the thread on a ready queue of runnable threads at the given priority level. A *lazy* allocation scheme delays stack allocation until the thread begins to run. This helps reduce memory requirements in applications where many threads are created, but only a few threads are active. A thread is 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 requirements. Threads are allowed a variable number of integer, float, and double type arguments. On a SPARCstation 20, the cost of thread creation is roughly 35 microseconds.

Context switching in Ariadne occurs on the stack of a thread yielding CPU control, with the aid of Unix’s `setjmp()` and `longjmp()` calls. A yielding thread saves its context in the *tca* area by invoking `setjmp()`. A scheduling function enables selection of the next runnable thread and gives it control via an invocation of `longjmp()`. When a `longjmp()` is called with an appropriate saved context as parameter, the corresponding thread resumes execution at the point where it invoked `setjmp()`. The cost of context switching between Ariadne threads is about 15 microseconds, on a

¹The system is currently being interfaced with kernel-space threads, where available, to combine the advantages of cheap, user-space threads with nonblocking execution. In the current system, use of multiple processes helps circumvent the problem of blocking.

SPARCstation 20. Details on context switching can be found in [25]. The approach is simple and effective, and has enabled ports to a number of different architectures, including SPARCs, SGI and IBM RS/6000 workstations, and Intel Paragon and Sequent Symmetry multiprocessors.

Scheduling and Time-Slicing

A built-in scheduler doles out portions of a process's time-slice to the process's threads. Each thread runs at one of several (integer) priorities, with FIFO scheduling within each priority class. At any given time, the highest priority runnable thread executes. It continues to run until it terminates, completes a time-slice (if the system is in time-slicing mode) or suspends execution.

In using a customized scheduler, Ariadne's kernel makes up-calls to user functions to acquire the next runnable thread; if a running thread blocks or suspends execution within a thread primitive, the thread is returned to the user via another up-call. User-defined functions typically create and manage an appropriate data structure for handling ready threads. This depends on how the application wants to schedule threads. Details of scheduler customization can be found in [23, 25].

Because threads are preemptable, all runnable threads at the highest priority have access to the CPU. This allows Ariadne threads to use fine time-slicing to share a host process's time-slice in round-robin fashion. For example, an application can use a receive-thread to receive data from, and a send-thread to send data to the network. A thread may adapt its time-slice to network or other load. This also offers the potential for multiplexing between I/O and computations, or between distinct types of computations, thus providing multiple virtual processors. Another use of time-slicing is in client-server applications. For example, a new server thread may be created for servicing each new connecting client. If multiple active server threads run within a time-slice, all clients may obtain service during the time-slice.

2.2 Threads Programming Primitives

Ariadne's basic sequential programming primitives are summarized in Table 1. The user-interface is simple. A user initializes the threads system through use of the `ariadne()` primitive. Once initialized, the invoking application is transformed into a "main" thread, and all Ariadne primitives are available to the application. After system initialization, the user typically creates multiple threads and uses primitives to perform various tasks. Use of `a_exit()` guarantees threads system termination, provided all user-created threads have either terminated or been destroyed. The application continues to run with a single thread of control from this point.

Control functions allow threads to suspend execution and yield control. Threads can be created with `a_create()` and destroyed with `a_destroy()`. Explicit transfer of control to a specific thread is done using the `a_yield()` function. The function `a_yieldc()` transfers control to the next thread at the same priority level, forcing a scheduler invocation. The `a_setpri()` call changes the priority of the invoking thread or a target thread, which may result in a context switch. The function `a_sleep()` allows a thread to sleep for a specified time.

Several query functions offer access to thread information. The function `a_ping()` indicates whether a specific thread is alive, and `a_self()` returns to a caller its own `thread_t` structure. The

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_mypr();	int a_myid();
void a_self(thread_t *thptr);	int a_ping(thread_t th);
void a_set_attrb(int index, void* buf, int sz);	void a_get_attrb(int index, void* buf, int sz);
Synchronization	
sem* a_creates(int value);	void a_signalalls(sem *semptr);
void a_signals(sem *semptr);	void a_signalns(sem *semptr, int count);
void a_waits(sem *semptr);	int a_counts(sem *semptr);
Mutual Exclusion	
a_mutex* a_mutex_create(a_mutex *mup, int p, int m);	
int a_mutex_lock(a_mutex *mup);	int a_mutex_unlock(a_mutex *mup);
int a_mutex_trylock(a_mutex *mup);	int a_mutex_destroy(a_mutex *mup);

Table 1: Threads programming interface

thread_t structure contains the thread identifier and a *tca* pointer, and is a parameter in Ariadne function calls. The *a_set_attr()* and *a_get_attr()* functions allow a thread to access its private data area.

Synchronization in Ariadne

Ariadne enables threads to synchronize through a simple set of primitives. Counting semaphores and mutual exclusion locks have been found to be adequate for most applications. The current implementation of semaphores allows creation of a semaphore, waiting on a semaphore with *a_waits()*, and signaling of a semaphore with *a_signals()*. These actions are similar to the **P** and **V** synchronization primitives on an OS. Function *a_signalalls()* signals all threads waiting on a semaphore, and *a_signalns()* signals a semaphore *n* times. Function *a_counts()* returns a semaphore count and may be used to determine whether a call to *a_waits()* will block.

Semaphores and locks are useful on both uni- and multiprocessors. On shared memory multiprocessors, they enable threads within distinct processes to synchronize. In distributed environments, message passing and thread migration are the primary synchronization mechanisms. A thread may migrate to a remote processor and signal a semaphore there. Details on basic kernel mechanisms, and benchmark comparisons of Ariadne primitives with primitives in commercial threads systems can be found in [25].

2.3 Parallel Programming with Ariadne

The basic threads system is augmented to provide programming support on shared and distributed memory environments. On shared memory multiprocessors, Ariadne exploits Unix's IPC facilities. Support is provided for the creation of multiple processes, shared memory access, installation of customized shared memory schedulers and thread migration. Distinct processes may execute distinct but shared threads, emulating kernel-space support. With scheduler data structures in shared memory and accessible to all processes, load-balanced execution of threads is automatic. Though processes compete for access to shared data structures, contention is reduced through use of locks with appropriate granularity, so that objects are locked only for small time intervals. Ariadne provides a simple interface for the allocation of user-defined data structures in shared memory, with concurrency control mechanisms for their access.

In distributed memory environments, such as workstation clusters of uni- and/or multiprocessors, Ariadne exploits existing communication infrastructures for thread management and migration. For example, Ariadne may be interfaced with PVM for workstation clusters, or with Intel communication primitives for the Paragon mesh. Coupled with a thread migration and an object location mechanism, basic threads primitives offer applications a powerful lightweight-process oriented parallel programming facility. Important factors that can hinder performance in such a scenario include poor data decomposition and mapping, frequent remote data access, frequent inter-process synchronization, and processor load imbalance. Issues of remote data access and synchronization are effectively addressed by thread migration and message passing. Migration and data redistribution also help in achieving reduced load-imbalance. Ariadne is highly effective with remote memory operations, active-messaging

Figure 2: Ariadne's shared memory model

and lightweight protocol design [16].

3 Ariadne Threads on Shared Memory

Since Ariadne does not require kernel-level support from an OS, true concurrency is achieved by multiplexing user-space threads on multiple processes. An alternative, currently being pursued, is to multiplex Ariadne threads on available kernel-space threads. The user can specify the number of processes to be created, typically equal to the number of available processors. Processes retrieve runnable threads from a shared queue or from a private queue before running these threads. On a blocking call, a thread's host is forced to block and is prevented from running other threads. Thus, having more processes than processors helps increase processor utilization. When a process blocks on a thread, the OS switches control to a waiting process which is free to run a ready thread. This is similar to kernel traps which occur in context-switches of kernel-space threads.

3.1 The Shared Memory Model

Ariadne's threads operate within processes on shared memory, as shown in Figure 2. Primitives supporting shared memory operations are given in Table 2. Upon initialization, a main process uses `a_shared_begin()` to fork off a user-specified number (`nprocs`) of processes, each of which has access to shared memory segments. In Figure 2, a main process forks off two children. Each process is given a shared process identifier `sh_process_id`, runs independently of other processes and has its own notion of a `main()` and a currently active thread. After a call to `a_shared_begin()`, a process invokes `ariadne()` to initialize the threads system. All thread primitives now become available for

Environment setup and exit	
<code>void a_shared_begin(int nprocs);</code>	<code>void a_shared_exit();</code>
Shared Memory Creation and Destruction	
<code>void* shmemalloc(int size);</code>	<code>void shmemfree(void*ptr, int size);</code>
<code>void* shmcreate(int key, int size, int perms, int *id);</code>	<code>shmfree(void* ptr, int id);</code>
Synchronization	
<code>int sem_id = a_createsh(int sem_key, int value);</code>	<code>int a_destsh(int sem_id);</code>
<code>int a_signalsh(int sem_id);</code>	<code>int a_signalnsh(int sem_id, int count);</code>
<code>int a_signalallsh(int sem_id);</code>	<code>int a_waitsh(int sem_id);</code>
<code>int a_countsh(int sem_id);</code>	
Scheduler Customization	
<code>struct tca* delete_ready(void);</code>	<code>struct tca* delete_named(struct tca* th);</code>
<code>void insert_ready(struct tca* th);</code>	<code>void insert_top(struct tca* th);</code>
<code>void a_sched.register(insert_ready, insert_top, delete_ready, delete_named);</code>	

Table 2: Ariadne's shared memory primitives

its use, so that a subsequent call to `a_exit()` ensures termination of all threads upon return. Function `a_shared_end()` in `main()` indicates termination of children and shared memory operation, so that only the parent process continues execution from this point. A template of a typical shared memory Ariadne program is shown in Figure 3.

Ariadne provides a simple interface to Unix's shared memory management primitives. A special *primary* shared memory segment of size `sz`, created during system initialization, is used for allocation of `tcas` and stacks at runtime. The number of shared threads that can be created is limited by the size of this shared segment. The primary shared memory is managed by the primitives `shmemalloc()` and `shmemfree()`, and is similar to heap space management with `malloc()/free()` in Unix. A user may also create and delete additional shared memory segments with `shmcreate()` and `shmfree()`. The cost of creating additional shared segments is an order of magnitude higher than the cost of `shmemalloc()`. The latter obtains memory from the existing primary shared segment, making it suitable for small amounts of shared memory or for creating/deleting chunks of shared memory frequently.

Synchronization primitives used with the shared memory system are semaphores and mutexes. The interface for mutex locks is the same as in the uniprocessor case; a parameter in the `a_mutex_create()` call indicates whether the mutex is used with a uniprocessor or a multiprocessor. Shared semaphores are identified by unique integers, and come with queues located in shared memory. A thread blocked on a `a_waitsh()` call in one process may be signaled via a `a_signalsh()` by a thread in another process, causing the blocked thread to be returned to a shared ready queue.

```

main()
{
    a_shared_begin(numprocs, sh_seg_size); /* create multiple processes */
    create_shared_scheduler();

    ariadne(&main_t, main_prio, base_stack_sz); /* initialize ariadne threads*/

    /* Ariadne's shared memory threads can now run */

    a_exit(); /* single threaded from this point */
    destroy_shared_scheduler();
    a_shared_end(); /* terminate processes */
    exit(0);
}

```

Figure 3: Shared memory program template

Ariadne threads may execute on a shared memory system in one of two mutually exclusive modes. With installation of an external shared memory scheduler, threads are allocated from a primary shared memory segment and are executable by any process. Without a customized scheduler, threads created within a process are permanently bound to the process. Threads bound to a process, however, can access shared data structures and thus cooperate with threads executing on other processes. Ongoing work with Ariadne seeks to permit mixed mode execution of shared and bound threads within a single application.

Shared Memory Schedulers

The base system provides a scheduler customization facility for implementing specialized shared memory schedulers. These allow processes to access and schedule threads from a common pool of runnable threads. User-defined customization involves development of data structure with insert and delete operations, callable by Ariadne's context-switching routine [25]. When invoked, the function `a_sched_register()` installs user-defined scheduler functions in place of the kernel's built-in scheduler. Given a sufficient number of runnable threads to keep processors busy, this mechanism effects automatic load balancing. Examples of load balancing with shared memory schedulers are described in Section 6.

With no threads to run, a process either remains blocked on a scheduler invocation or is returned a NULL pointer. In the latter case, Ariadne's kernel schedules an internal bridge thread. This thread spins, waiting for new threads to be created, for threads asleep on a delta queue to awaken, or threads blocked on a semaphore to be signaled. Ariadne allows either blocking or spinning to be

used, depending on application needs. With blocking, the OS is free to schedule another ready process, improving overall system utilization. With spinning, useful CPU cycles are consumed, though overall performance improves because expensive OS-level context switches are avoided.

Performance considerations

Many inherently concurrent applications can exploit shared memory multiprocessing, and Ariadne is well-suited to these. For example, parallel simulation or event-based applications may use a centralized data structure to store events in shared memory, allowing threads in distinct processes access to events. Though virtually any concurrent application may run processes sharing Ariadne's threads in this manner, performance is best when loads are balanced. If load is measured in terms of the number of ready threads, load-balancing is done by making each processor execute roughly the same number of threads. While using threads in large numbers sometimes simplifies code development and helps reduce load imbalance, the cost of thread creation and destruction, relative to the work done by a thread, can offset potential performance gains. As will be shown with a multithreaded Quicksort in Section 6, it is better to sort small lists using a function instead of a new thread. Through the same example, it is shown that threads with balanced workloads yield better performance than threads with unbalanced workloads.

Conflicts occurring during shared memory access can degrade performance significantly. An example of multithreaded adaptive quadrature, given in Section 6, demonstrates this point. The adaptive quadrature involves a fine-grained computation, requiring every thread that computes the value of a function over a given interval to update a value in shared memory. To reduce memory conflict and improve performance, threads temporarily store results in a local area. A low priority "update" thread, executing only when no "compute" threads are available, transfers results from the local to the shared area. The use of the "update" thread is not necessary for most situations; it is only a simple and effective way to reduce contention and improve performance.

On shared memory, Ariadne's threads are well-suited to medium and coarse grained operations obtained by agglomeration. Many threads may coexist in the system, requiring substantial shared memory space. A serious constraint here is the size of shared memory segments, usually configured small by system administrators. This can sharply curtail creation of multiple shared threads. Reconfiguring a system to allow larger shared segments is simple. In the extreme, however, if an application requires more memory than is available on a single processor or multiprocessor, there is little recourse but to use a distributed cluster of processors.

4 Ariadne Threads on Distributed Memory

Principal factors motivating the development of parallel applications on distributed systems include large problem sizes and the need for improved performance. Indeed, though a number of support-level applications (e.g., collaborative computing, network protocols) make sense only in distributed settings, performance is always a prime consideration. Both with end-user applications as well as support-level applications, a distributed threads system can significantly enhance a system's functionality, improve

Figure 4: Ariadne's distributed memory model

performance – in terms of reduced execution time and development effort, and increased expandability – and ultimately offer a variety of operational paradigms not easily had in traditional process-based systems. Ariadne provides such a unified threads framework atop a virtual multiprocessor, enabling very high levels of concurrency.

As an example of an operational paradigm, distributed threads offer a natural environment for supporting active objects on static or dynamic data domains. For example, in the study of percolation phenomena, threads are natural candidates for the representation of moving particles. A domain is typically split into distinct objects which are divided among processors. Issues of data locality will arise in such settings, and these are effectively addressed by thread migration. In the study of network design or mobile computing systems, threads may represent packets or dynamic computing units. Besides computational or modeling applications, distributed threads are valuable in support-level settings such as enhanced windowing, active-messaging and protocol design [16]. In general, distributed threads perform well when computation granularity is coarse. Distributed applications described in Section 6 include a distributed successive over-relaxation (SOR) linear solver, a particle-physics application, and adaptive quadrature.

4.1 The Distributed Memory Model

A layout depicting Ariadne's operation on distributed memory is shown in Figure 4. The system initializes one or more distributed processes – henceforth called D-processes – on each processor. Each D-process is given a unique integer id starting at 0. A D-process may fork off children on a uni- or multiprocessor and handles all communication between itself (and/or its children) and other hosts in the system. Child processes are identified with an integer-tuple $\langle \text{process identifier}, \text{shared process identifier} \rangle$. In Figure 4, three D-processes are shown: processes 0 and 2 are located on distinct multiprocessors, and process 1 is located on a uni-processor. On each multiprocessor, the

D-process forks off children; parents and children share a common threads pool. The uniprocessor hosts a single D-process.

Ariadne's D-processes communicate via messages, using primitives from an underlying communications library (PVM, Conch, etc). Child processes interact with their parents via shared-memory, resulting in significant cost savings, since shared-memory IPC is cheaper than socket-based IPC. Ariadne's D-processes may either be dedicated to communication or share computational load with children, the choice depending on the message density of a given application. As shown in Figure 4, the distributed computing model 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). D-processes encapsulate Ariadne threads and global objects.

Threads are free to move between any D-processes. In Figure 4, a thread migrates from D-process 0 to D-process 2. Typically, threads move to access global objects located at other processes, as computations that chase data. Global data objects are generally static. To effect thread-migration, Ariadne utilizes an object locator to locate the right target host it must migrate to for accessing data.

Object Location

Implementing an object-locator is fairly straightforward. Consider, for example, the emulation of particle movement on a 2-D grid: horizontal (or vertical) slices of the grid, treated as global objects, are distributed across distinct processes. An elementary calculation maps grid locations to global objects. When a particle (thread) traversing a grid slice (object) crosses a boundary, it must access another grid slice (object). If the target object resides on another process, the thread must migrate. After migration, it continues to execute on the target process until further locality constraints force it to move on.

How global objects are assigned to processes depends on the application and the user: the key is to minimize communication. Good techniques for minimizing global state distribution is important – to enable large granularity in, and infrequent migration between computations. It helps to keep migration costs small relative to computation times.

Programming with distributed threads

A list of Ariadne functions, useful in distributed settings, is shown in Table 3. With help from the underlying system, these enable process creation and termination. For example, PVM allows process hosts to be specified interactively; with Conch, a host file is required. When invoked, the function `a_dinit()` initializes the distributed environment, and `a_dexit()` triggers a graceful shutdown. Query functions `a_getprocid()` and `a_getnumprocs()` return a process identifier and the total number of processes, respectively. Function `a_printf()` enables writes, useful in debugging distributed applications. For example, Conch allows writes to a front-end processor (i.e., a processor initializing the Conch system), and PVM allows logging.

Crucial in distributed settings are functions for message passing, thread migration and termination

Function	Description
a_dinit(void)	Initializes the distributed environment
a_dexit(void)	Shutdown of system/processes
a_getprocid(void)	Returns the process identifier of the process
a_getnumprocs(void)	Returns the number of processes in the system
a_printf(char* format, ...)	Remote message printing
a_terminate_check(void)	Check for distributed termination
a_migrate(int procid)	Migrate the calling thread to procid
a_spawn(int procid, func, ...)	Creates a thread at a remote process
a_register(char* th_name, func)	Register a function at this process with a name
a_exec(char* th_name, ...)	Execute a registered function as a thread or a function
a_probe_messages(void)	Receive and handle a thread migration, destruction, or process termination message

Table 3: Ariadne's distributed memory primitives

```

main()
{
    a_dinit();                /* initialize distributed system
                             and multiple processes */
    numprocs = a_getnumprocs();
    myprocid = a_getprocid();

    /* initialize ariadne threads*/
    ariadne(&main_t, main_prio, base_stack_sz);
    a_usr_cswitch_proc = sp_process; /* check for migrated threads
                                     * after each context switch */

    /* Ariadne's distributed threads can now run */

    a_exit();                 /* single threaded from this point */
    a_dexit();                /* terminate processes */
    exit(0);
}

```

Figure 5: Distributed memory program template

detection. Communication primitives (for example, `pvm_send()` and `pvm_recv()` in PVM) may be employed to send and receive data. Ariadne enables thread-migration via the `a_migrate()` primitive. Function `a_probe_messages()` checks for Ariadne's system messages, including migrant threads and messages for destroying threads or terminating processes. Function `a_dterminate_check()` checks for distributed system termination, and function `a_spawn()` enables creation and execution of threads on remote processors. The latter is ideal for redistributing load. Function `a_register()` registers a function globally with all processes, using a unique (user-specified) name, executable using `a_exec()`. This primitive has an option which indicates whether the given target should be invoked as a function or as a thread.

A typical program template, demonstrating use of Ariadne's distributed memory primitives, is shown in Figure 5. Each distributed process is addressed by the variable `myprocid`. Observe that the shared and distributed memory versions of an Ariadne program are much alike. Only function `main()` changes; all other thread-functions undergo little or no change. For example, the shared memory call `a_shared_begin()` is replaced with the distributed memory call `a_dinit()`. This point is made clear with the adaptive quadrature example given in Section 6, showing code for the sequential, shared- and distributed-memory cases.

4.2 Thread Migration

Ariadne provides direct support for two message-based computing models: threads-based `send()/recv()` and thread migration. Both rely on the underlying communications environment and may be generalized to envelop other computing models. Using message-passing, for example, Ariadne processes may support remote-memory copy; through data area broadcasts, remote processes are offered asynchronous read/write access. As another example, mobile threads in the ParaSol [22] parallel simulation system may communicate with other mobile threads through a mailbox facility. Such messaging enables threads to exchange and share data, update results, schedule computations and synchronize. Thread migration overheads in Ariadne are small, compared to the total time required for migration. The time required to migrate a thread is only 1-3% larger than the time required to move a message of the same size. Details on thread migrations can be found in [25].

Each RPC-style [5, 9, 11] access requires two messages, a `send` and a `receive`. The former delivers parameters to a procedure, and the latter returns results. In the traditional process-oriented model the sender typically blocks on the `send`. With threads, the sending process is free to continue with other work. Data-shipping is a form of remote memory copy: remote data is moved to a computation or simply moved from one process to another [20, 11]. Shipping data to a computation requires copy coherency, an expensive proposition when changes are frequent. Moving data between processes to improve locality is beneficial when remote accesses are frequent, but is expensive for large data blocks or memory pages. Complex object migration is nontrivial, given that internal objects and/or pointers to other objects require packing and unpacking at source and destination processes, respectively.

Another alternative to the two schemes mentioned above is computation migration [18, 8]. Here, part of a computation (e.g., the topmost function in a sequence of nested calls) is moved to a process hosting required data. Using compile-time transforms, the scheme described in [18] effects this by

transferring single activation frames between hosts; the function is forced to return to the sender, entailing a two-way transfer. In the Ariadne system a thread is free to move, along with all its activation frames, to access data in another processor; it is not required to return to the sender. Local objects present on the thread's runtime stack also move with the thread. Programming with thread migration is similar to programming in the shared memory model, because all data accesses become local after migration. Because a migrant need never return to its sender, transfers are one-way instead of two-way; intermediate results are generally stored in an object within the thread.

Thread migration is known to require the least number of messages for a typical series of remote access requests [18]. With potentially low consumption of network bandwidth, relative to other access schemes, thread migration offers good potential for improved performance. Choice of an appropriate access scheme depends, however, on the needs of the application and on the underlying environment. Hybrid schemes may work well, because many applications exhibit a variety of data-access characteristics. Frequent thread access to large, remote objects is best replaced by migrating the thread to the object's host. Frequent thread migration from one host to another, for access to a small object, is best replaced by moving the object to the thread's host. Data migration will perform poorly with replicated write-shared data, or when data objects are large. Thread migration will perform poorly if a thread's state is large relative to the granularity of computation between consecutive migrations. There are several points worth noting with regard to thread migration:

- *Post-migration code is similar to shared memory code.* In the pseudocode shown below, the variable `sum` is directly accessible to a thread once this thread arrives at the process hosting this object. The `isRemote()` test is not essential – the thread returns from the `a_migrate()` call if the object is local.

```
Object *pObj;                /* a global object */

void updateSum(int sum)
{
    if (pObj->isRemote())
        a_migrate(pObj->location());
    pObj->sum += sum;          /* sum is local after migration */
}
```

- *Thread Migration enables data to migrate with computations.* Data present on a thread's stack moves along with the thread. This offers a mechanism for message-passing. Data essential to the thread's continued execution on a target host may also move with the thread. In the pseudocode shown below, a thread takes a random number seed along with it on its travels. The thread may use this seed to generate one or more random numbers on a target host, updating the seed with every generation.

```

Object *pObj;

void customer(int id)
{
    int seed = INITSEED;
    int randomNo;

    if (pObj->isRemote())
        a_migrate(pObj->location());
    randomNo = rand(&seed);    /* seed is updated after migration */
    pObj->updateValue(randomNo);
}

```

- *Thread Migration enhances locality of access.* When a thread requires access to multiple objects or multiple accesses to a single object on a remote host, thread migration wins over other schemes, because only one message transmission is required.
- *Thread Migration enables synchronization.* Between computation phases, synchronizing threads may migrate between processors to signal semaphores.
- *Thread Migration can effect balanced loads.* When threads are not bound to hosts because of data requirements, they may migrate to lightly loaded hosts to balance load. When bound, a combination of thread and data-object migration can be used to reduce load-imbalance.
- *Thread migration aids in data-driven computation.* Because threads may be scheduled by an application, immigrant threads can be made to run at select times. In particular, such threads may run only when data dependencies are satisfied. Clever use of the scheduling capability, in tandem with migration and data-dependency, will result in efficient data-driven execution. Immigrant threads will run only when required, minimizing creation and context-switching overheads.

Thread Migration and RPC

There may be instances in which a remote procedure call like (RPC-like) thread invocation is cheaper than thread migration. This occurs, for example, when one or more of the following is true: the current thread requires access to a remote computation, the computation stack is large, and repeated accesses are not required. Instead of migrating a thread, it is possible to spawn off a new thread remotely, using the `a_spawn()` primitive. The thread context information sent to the remote site need only contain a function pointer and function arguments. Once this information is sent, the spawned thread executes at the remote site, and the sender is free to continue processing other threads. The sender waits on a local semaphore, when and if the result of the spawned operation is required. On completing the remote operation, the spawned thread migrates back to the sender, to signal the semaphore. The returning thread need not contain all the activation frames used by the spawned thread. The C code segment shown below explains how this is done.

```

sem *pSem;                                /* synchronizing semaphore */

void client_thread(...)                    /* on processor procC */
{
    int val;

    a_spawn(procS, func, PRIO, TINY, 1,0,0, &val);

    /* other processing while func computes value */

    a_waits(pSem);                        /* will yield to other ready threads */
    a_printf('`val = %d`', val);
}

void func(int *pVal)                      /* on processor procS */
{
    int ret;

    ret = computeVal();
    a_migrate(procC);                    /* procC is where the client is running */
    *pVal = ret;                         /* this executes on procC */
    a_signals(pSem);
}

```

5 Customization Interface

The base system may be interfaced with any communications subsystem, such as PVM, Conch, P4, Intel communication library etc. In this section customization with PVM is used to demonstrate important features of Ariadne's lower layer support, including core facilities for remote process creation/termination and traditional process-based message passing. For example, PVM provides a `pvm_spawn()` / `pvm_exit()` primitive for creation/termination of remote processes, and `pvm_send()` / `pvm_recv()` primitives for communication. The functions described in Table 3 enable customization. Given below is a brief description of how system initialization and thread migration are accomplished with the help of `a_dinit()` / `a_dexit()` and `a_migrate()` primitives.

Implementation of `a_dinit()` and `a_dexit()`

Function `a_dinit()` uses the process creation facility offered by the communication subsystem to create remote processes and set up a distributed environment. Each process is given an identifier (PVM uses system-based task identifiers which are mapped into integers 0, 1, 2, ... by Ariadne), obtainable via the `a_getprocid()` call. During initialization, the first process created turns into a "control" process which monitors progress of other processes; it determines when all processes are done with computation, and can trigger a shutdown of the system. Details on how this is accomplished can be found in [25]. Function `a_dexit()` effects cleanup on process termination, using primitives provided

by the underlying system for this purpose. For example, in Ariadne's PVM interface, a call is made to `pvm_exit()`. Pseudocode outlining this implementation is shown in Figure 6.

Asynchronous Message Handling

Ariadne provides three internal message types: termination (FINISH/EXIT), thread destruction (THDEST), and thread migration (THMIGR). The control process sends the application a termination message (EXIT) when all user-created threads have terminated. Upon receipt of such a message, the application may invoke termination routine `a_set_end()` in the support layer. A process sends a THDEST message to a thread's creator (process) when it destroys a thread, and sends a FINISH message to the control process when it has no more threads to run. A THMIGR message contains a thread's snapshot. This snapshot can be unpacked and turned into a live thread at a receiving host. A user-written handler is invoked to handle each type of message at a receiver. The nature of the handler defines Ariadne's use for a given application. Message types and handlers are available via a function, invocable after every context switch. This is accomplished by making Ariadne's internal variable `a_usr_cswitch_process` point to the message handling function.

Implementation of `a_migrate()`

A PVM based implementation of `a_migrate()` is shown in Figure 7. The thread-migration interface consists of two support layer primitives: `a_thread_pack()` and `a_thread_unpack()`. The former is invoked by a source process to dissect and pack a thread into a buffer, to be sent over a network. The latter is used by a target process, upon message receipt, to transform a message into a live thread. Processors are assumed to be homogeneous, so that data translation involving byte ordering of data on the stack is unnecessary. Once a thread is packed and sent, the sender destroys its own copy of the thread, so that the thread ceases to exist at the sender.

6 Examples

We present four examples of parallel programs developed using Ariadne. Differing in application type, computation granularity, type of data access (shared or remote), and number of working threads, these examples demonstrate Ariadne's versatility for parallel programming. In the first example, adaptive quadrature is used to estimate the value of an integral on a given domain. Sequential, shared- and distributed-memory versions of code are presented. The second example shows how a quicksort algorithm can be implemented with threads, with choice of a pivot element shown to impact runtime performance. Two different pivots are compared: the first element, and the median of the subarray. The third and fourth examples implement coarse-grained applications on distributed memory. In the third example, static threads update grid points in distributed successive over-relaxation (SOR), a template-based scheme for solving linear equations. In the fourth example, dynamic threads model particle movement in an application of polymer physics.

```

int numprocs;                /* number of processes in the system */
int procid;                  /* process identifier for this process */
int tids[MAXPROCS];

/* a_dinit() sets up process initialization and termination data structures */
void a_dinit(char **argv, int nprocs)
{
    int mytid;
    int done = FALSE;

    numprocs = nprocs;
    mytid = pvm_mytid();      /* pvm routine to get task id */
    tids[0] = pvm_parent();
    if (tids[0] < 0) {        /* the controller */
        tids[0] = mytid;
        procid = 0;

        pvm_spawn("programe", &argv[1], 0, "", nprocs-1, &tids[1]);
        pvm_initself, pvm_pkint; /* init and pack the tid array */
        pvm_mcast(&tids[1], nprocs-1, 0); /* send to others */

        set up data structures for termination since I am controller;

        while (!done) {
            recv FINISH messages from other processes;
            done = FALSE;
            if (all processes have finished)
                done = TRUE;
        }
        send EXIT message to all processes;
        pvm_exit();
    }
    else {
        pvm_rcv and pvm_upkint tids array; /* receive and unpack */
        procid = value of index based on tid array;
    }
}

/* a_dexit() is called to exit the pvm system; a_exit() is already done */
void a_dexit()
{
    pvm_exit();
}

```

Figure 6: Implementation of a_dinit() and a_dexit() using PVM

```

void a_migrate(int procid)      /* migrate a thread */
{
    int jump;                  /* distinguishes between source and destination */
    char *image;               /* thread image is placed here */
    int size;                  /* size of the packed thread */
    thread_t self;

    jump = a_thread_pack(0, &image, &size);
    /* image now contains the thread tca and stack */
    if (!jump) {               /* we are at the source */
        a_set_migrate(0);
        pvm_initsend(PvmDataRaw);

        pvm_pkbyte(th_stack_ptr, size, 1); /* pack to form a message */
        pvm_send(tids[procid], THMIGR); /* send the thread to procid */
        a_add_migrated();             /* update counters */
        free(image);                 /* free dead image */
        a_self(self);                /* get identifier */
        a_destroy(self);             /* destroy thread on source */
    }
    else
        a_reset_migrate(0);         /* we are at the destination */
}

```

Figure 7: Implementation of `a_migrate()` using PVM

6.1 Adaptive Quadrature

Many classical schemes exist for numerical integration, e.g., the rectangular, midpoint, trapezoidal, and Simpson's rules. In each case the domain of integration is subdivided into equal-sized subintervals. Numerical integration is performed on each subinterval and the results combined to yield the required estimate. Adaptive quadrature is a technique which allows more flexibility. The domain is subdivided into only as many subintervals as are necessary to obtain a given accuracy. Further, subintervals may differ in size. In the one-dimensional version, the problem is to compute an estimate P of the integral

$$I = \int_a^b f(x)dx$$

so that $|P - I| < \delta$, for a small and positive δ .

A sequential algorithm for computing P is shown in Figure 8. Function `f`, endpoints of interval $[a, b]$, and tolerance δ , or `tol`, are inputs. Integration is based on Simpson's approximation rule. Function `aquad()` implements the algorithm recursively: a recursive call is made if the estimate on a subinterval exhibits an error greater than the δ -value specified for that subinterval. Each recursive call involves computation of `f` at two new points in the subinterval. Error is computed as a function of the previous and current estimates.

The sequential code is readily adapted to a parallel version, based on the shared memory model. But without recursion, a new thread is created for computation over each new subinterval. A C function

```

/*
 * f - the integrand
 * [a, b] - the interval
 * tol - the desired accuracy
 */
double est; /* the result is stored here */

main(f, a, b, tol)
{
    c = (a+b)/2; /* interval midpoint */
    fa = f(a), fb = f(b), fc = f(c); /* compute function at a, b, c */
    p = Simpson_rule(a, c, b, fa, fc, fb); /* current estimate */

    aquad(f, a, c, b, fa, fc, fb, p, tol);
    printf("The estimated value is %g\n", est);
}

aquad(f, a, c, b, fa, fc, fb, p, tol)
{
    c1 = (a+c)/2;
    c2 = (c+b)/2;

    fc1 = f(c1), fc2 = f(c2);
    s1 = Simpson_rule(a, c1, c, fa, fc1, fc);
    s2 = Simpson_rule(c, c2, b, fc, fc2, fb);
    p1 = s1 + s2;

    error = compute_error(p1, p); /* error in p1 */
    if (error < tol)
        est = est + p1;
    else {
        aquad(f, a, c1, c, fa, fc1, fc, s1, tol/2);
        aquad(f, c, c2, b, fc, fc2, fb, s2, tol/2);
    }
}

```

Figure 8: A sequential adaptive quadrature algorithm

```

struct user_data {                /* shared area data structure */
    double estp;                  /* the result is stored here */
    a_mutex *lock;
};

struct user_data *usdp;           /* shared area */

aquad_norecur(usdp, f, a, c, b, fa, fc, fb, p, tol)
{
    c1 = (a+c)/2;
    c2 = (c+b)/2;

    fc1 = f(c1), fc2 = f(c2);
    s1 = Simpson_rule(a, c1, c, fa, fc1, fc);
    s2 = Simpson_rule(c, c2, b, fc, fc2, fb);
    p1 = s1 + s2;

    error = compute_error(p1, p); /* error in p1 */
    if (error < tol) {
        a_mutex_lock(usdp->lock); /* lock for update */
        usdp->estp = usdp->estp + p1;
        a_mutex_unlock(usdp->lock); /* release lock */
    }
    else {
        a_create(0, aquad_par, 5, MEDIUM, 2, 0, 8,
            usdp, f, a, c1, c, fa, fc1, fc, s1, tol/2);
        a_create(0, aquad_par, 5, MEDIUM, 2, 0, 8,
            usdp, f, c, c2, b, fc, fc2, fb, s2, tol/2);
    }
}

```

Figure 9: Parallel adaptive quadrature on shared memory (recursion-free)

implementing this thread, called `aquad_norecur`, is shown in Figure 9. Because recursive calls are replaced by thread creations, many threads must be created to replace deep recursive calls. Function `main()` closely follows the template shown in Figure 3, and is not repeated here. The parent process allocates a shared memory area from Ariadne's internal shared segment to store the result `estp`; it creates the first thread, which in turn creates other threads. Because distinct threads update `estp` simultaneously, a mutex lock guards access. This lock is created using the `a_mutex_create()` primitive. Several improvements are possible, even while retaining the simplicity of the initial version.

Performance of the scheme just described is poor primarily because thread creation costs outweigh thread work. That is, thread `aquad_norecur()`'s computation is too fine-grained and does not fully justify its creation. Also, conflicts in shared value updates cause some serialization. Because many threads with approximately equal work are created, load imbalance is not a problem. A single queue stores work to be done in the form of runnable threads. Upon becoming free to do work, a process acquires a thread from the head of the queue. Threads created by one process may be run by another process, making thread-migration between processes transparent.

A more efficient parallel version is shown in Figure 10. Here, two types of threads are used: a "compute" thread, and an "update" thread. The former is `aquad_recur()` which behaves as before but now updates results in a local area; so it does not need locks. The latter is an `update()` thread which uses a lock to move data from the local area into shared space. It is given low priority and runs only when no compute threads need to run. The inclusion of recursion in the new scheme helps reduce the number of threads created, thus reducing creation costs; the work done by each thread increases. New threads are created only when the depth of recursion reaches or exceeds some user-set threshold `RECURSE_LEVEL`. This helps prevent stack overflow and increases computation granularity. A related technique for increasing the granularity of fine-grained applications is found in Lazy Task Creation [26]. It is also possible to enhance stateful threads with support for low-cost fine-grained operations as in the Chores system [12].

An interesting feature of programming with Ariadne, as demonstrated above, is the relative ease with which one can move from sequential code to parallel code. This occurs because sequential and parallel versions of code do not differ much. Differences occur primarily in function `main()` (see templates in Figures 3 and 5). The functions `aquad()`, `aquad_recur()` and `aquad_norecur()` differ only in that `aquad()`'s recursive calls turn into thread creations in the parallel code. Moreover, Ariadne's basic primitives remain the same in the multiprocessor setting, with new primitives only enhancing parallel system support. Thus, transforming sequential applications into parallel applications generally requires little effort.

For completeness, we also provide a view of this problem in the distributed memory model. The code is shown in Figure 11. In the shared memory model's thread `aquad_recur()`, two new threads are created in the same host when recursion depth violates the user-set limit. To help maintain a balanced load, thread `aquad_dst()` in the distributed memory model simply *spawns* off one of these two threads on a remote processor. In our example, exactly which processor will receive the spawned thread is decided via a round-robin allocation scheme, based on processor `id`. A number of other schemes may be equally viable. Final results are computed by process 0, after it receives all partial results from other processes. Process 0 simply sums up all results it receives, including its own.

```

struct user_data {                /* shared area data structure */
    double estp;                  /* the result is stored here */
    a_mutex *lock;
};

struct user_data *usdp;           /* shared area */
double estp;                      /* local area */

aquad_recur(recursion, f, a, c, b, fa, fc, fb, p, tol)
{
    c1 = (a+c)/2;
    c2 = (c+b)/2;

    fc1 = f(c1), fc2 = f(c2);
    s1 = Simpson_rule(a, c1, c, fa, fc1, fc);
    s2 = Simpson_rule(c, c2, b, fc, fc2, fb);
    p1 = s1 + s2;

    error = compute_error(p1, p); /* error in p1 */
    if (error < tol)
        estp += p1;              /* lock-free update */
    else {
        if (recursion < RECURSE_LEVEL) { /* RECURSE_LEVEL is user-specified */
            aquad_shared2(++recursion, f, a, c1, c, fa, fc1, fc, s1, tol/2);
            aquad_shared2(++recursion, f, c, c2, b, fc, fc2, fb, s2, tol/2);
        }
        else {
            a_create(0, aquad_par, 5, MEDIUM, 2, 0, 8,
                     0, f, a, c1, c, fa, fc1, fc, s1, tol/2);
            a_create(0, aquad_par, 5, MEDIUM, 2, 0, 8,
                     0, f, c, c2, b, fc, fc2, fb, s2, tol/2);
        }
    }
}

void update(struct user_data* usdp, nprocs)
{
    do {
        a_mutex_lock(usdp->lock);
        usdp->estp += estp;
        estp = 0.0;
        a_mutex_unlock(usdp->lock);
    }
    while (a_numbthreads() < nprocs);
}

```

Figure 10: Parallel adaptive quadrature on shared memory (limited-recursion)

```

/* results area */
struct user_data {
    double aquad;
};

int numprocs, myprocid;
struct user_data usd;
typedef double (*fp)(double); /* function to be integrated for a <= x<= b */

void
aquad_dst(int recursion, fp f, double a, double c, double b,
          double fa, double fc, double fb, double p, double tol)
{
    c1 = (a+c)/2;
    c2 = (c+b)/2;

    fc1 = f(c1), fc2 = f(c2);
    s1 = Simpson_rule(a, c1, c, fa, fc1, fc);
    s2 = Simpson_rule(c, c2, b, fc, fc2, fb);
    p1 = s1 + s2;

    error = compute_error(p1, p);
    if (err < tol)
        integrt.aquad += aquad;
    else {
        if (recursion < RECURSE_LEVEL) {
            aquad_dst(++recursion, f, a, c1, c, fa, fc1, fc, s1, tol/2);
            aquad_dst(++recursion, f, c, c2, b, fc, fc2, fb, s2, tol/2);
        }
        else {
            dest = (myprocid+1)%numprocs;
            /* perform remote spawning to balance load */
            a_spawn(dest, aquad_dst, 4, HUGET, 2, 0, 8,
                    0, f, a, c1, c, fa, fc1, fc, s1, tol/2);
            a_create(0, aquad_dst, 4, HUGET, 2, 0, 8,
                    0, f, c, c2, b, fc, fc2, fb, s2, tol/2);
        }
    }
}

```

Figure 11: Adaptive quadrature on distributed memory (remote spawning)

Figure 12: Performance of adaptive quadrature

Performance

We tested the adaptive threads-based quadrature on an integrand proposed in [19] to be a worthwhile test. The function $f(x) = \text{sign}(g_1(x), g_2(x))$ is defined as $|g_1(x)|$ if $g_2(x) \geq 0$, and as $g_1(x)$ if $g_2(x) < 0$. For our experiments, $g_1(x) = x * x + 1.0$, $g_2(x) = \sin(x)$, with $x \in [-10, 10]$, and $\delta = 10^{-7}$. All runs were made on a 4-processor SPARCstation 20.

Measured execution time and speedup are shown in Figure 12. Measured time excludes initializations, and speedup is measured relative to sequential (single-threaded) execution. As explained earlier, the initial non-recursive threads version does not perform well because of the small computation granularity to creation cost ratio, and contention for shared variable access. Using threads with limited recursion yields significant improvement. Experiments limiting recursion depths to 50 and 150 show good performance. It is interesting to note that when recursion depth is unlimited, performance exceeds that of the non-recursive threaded version, but falls below that of the sequential and multi-threaded limited-recursion versions. As shown in the graphs, speedups of up to 3 were measured on a non-dedicated 4 processor machine.

In another interesting result, the uniprocessor shared memory version based on Ariadne threads turned out to be slightly *faster* than the sequential version when recursion is limited. This observation is significant. One would expect the reverse to be true, because of Ariadne's overheads. It turns out that the unlimited recursion in the sequential version causes severe page-faulting as recursion depth and stack size increases. Stacks as large as 4MB were observed with recursion depths of 19000. With Ariadne's threads, stack sizes are usually limited and so cache misses are not as frequent. This interesting phenomenon, however, warrants further study and is currently being investigated.

Some simple usage-related statistics on threads, for the quadrature problem, are given in Table 4. The

Program	Total Threads	Maximum Live Threads		
		Number of Processors		
		1	2	4
Non Recursive	480877	87	378	1038
Recursive Level 50	11718	61	250	288
Recursive Level 150	3987	63	139	200
Full Recursion	16 - 217	6	6	45

Table 4: Simple thread statistics on adaptive quadrature

total number of threads created (shown in the second column) is typically far greater than the maximum number of coexisting threads (shown in the third column, for different processor configurations). Both numbers drop when recursion is introduced. Fewer threads implies reduced threads management and context-switching costs, which is why performance improves. In the adaptive quadrature example, threads run to completion upon obtaining control. Before relinquishing control on termination, each thread may create two new threads. Hence the context-switch count is in the order of the number of threads.

6.2 Quicksort

The well-known quicksort algorithm [1] operates recursively, to sort a list S of n integers. A `partition()` procedure uses a pivot to split S and each recursively obtained subarray into three pieces. The first piece contains elements smaller than the pivot, the second contains elements equal to the pivot, and the third contains elements greater than the pivot. Once the pieces are defined, the quicksort is invoked recursively to operate on the first and third pieces. Since array partitioning is work enough, pivot selection is usually done as rapidly as possible. Finding pivots that generate balanced arrays requires extra work and is rarely used in quicksorts.

As in the previous example, the sequential algorithm readily lends itself to parallelization on shared memory using Ariadne's threads. As before, each recursive call that operates on a subarray of S is turned into a thread. Code describing the development of a parallel, multithreaded quicksort can be found in [25]. In the following section we show how pivoting to get balanced loads can enhance parallel execution performance, even though such pivoting requires more work.

Using Ariadne to perform a quicksort helps illustrate its automatically balanced processor loads. A single queue stores work to be done in the form of runnable threads. When a process is free to work it acquires a thread from the head of the queue. Threads created by one process may run on another. Unlike the quadrature example, synchronization mechanisms are not required. Each thread simply works on a distinct subarray, with its amount of work depending on the size of the subarray. Clearly, poorly chosen pivots can generate wildly different thread workloads. Recall that in the adaptive quadrature example, all threads did the same amount of work.

Figure 13: Performance of Quicksort

Performance

Experiments were conducted on a 4-processor SPARCstation 20. Measured performance of the multithreaded quicksort, on integer arrays of sizes ranging from 500000 to 4000000, is shown in Figure 13. Initialization time – memory allocation and reading S into memory – is ignored, and only the time required for performing the sort is reported. Because of thread-related overheads, it may be more effective to sort small arrays via function calls. For example, on a SPARCstation 20, it is cheaper to invoke a function in sorting a list of 110–200 instead of creating a thread. Because of this, when subarrays with less than 110 integers are generated, our quicksort routine simply invokes a bubble-sort procedure instead of creating new quicksort threads.

As shown in Figure 13, performance is good when sorts are done on large arrays. The number of threads created in total depends on the initial size of S : from 13,397 threads for a 500000-integer array to 94,466 threads for a 2000000-integer array. Processor loads are balanced because each process creates, runs and switches context between roughly the same number of threads. The uniprocessor, shared memory version of Ariadne takes about 12–15% longer than a sequential program, giving a rough indication of threads system overheads.

6.2.1 Quicksort with FIND

In early work, Hoare [17] proposes a FIND algorithm for locating the order-statistics of an array. The algorithm locates the j -th smallest array element through a scan which splits an N element array A in such a way that $A[0], A[1], \dots, A[j-2] \leq A[j-1] \leq A[j], \dots, A[N-1]$. FIND can be used to locate the *median* and halve an array. The idea is to give newly created quicksort threads equal

Figure 14: Quicksort with FIND

Number Of Integers	Number of Processors		
	1	2	4
500000	1.88	1.57	1.34
1000000	1.95	1.83	1.36
2000000	1.73	1.57	1.26
4000000	1.45	1.35	1.18

Table 5: Time(Quicksort)/Time(Quicksort+FIND) ratios

workloads. Because the recursion tree is balanced, concurrency is enhanced and fewer threads may be created. Newly created pairs of threads are given equal work, thus equalizing load and improving performance. It is interesting to observe that the FIND algorithm, whose average run time is twice that of a simple-minded partition, significantly enhances quicksort performance on a multiprocessor.

The FIND-based quicksort was run on a 4-processor SPARC 20, using the same parameters as in the original version. Execution times are shown in Figure 14, and ratios of the latter run times to the former run times are given in Table 5. The FIND-based quicksort shows a performance improvement ranging from 18% to 88%, primarily because fewer threads are created. Performance improvements decrease even as processors and array sizes increase, possibly because thread workload balance has less of an impact when process workload is already balanced and all processors are busy.

6.3 SOR Computation

Threads lend themselves well to iterative computations, such as to be found in linear system solvers. Such iteration occurs, for example, in determining steady state heat distribution over a thin square metal plate, given temperature on the boundary. The behavior of this system is governed by Laplace's equation with Dirichlet boundary conditions. It can be discretized using a five-point difference equation [30], with the square replaced by a mesh. The problem is one of determining the values of heat distribution at mesh intersection points.

Given a mesh with m^2 points, the finite differencing scheme yields a system of m^2 linear equations in m^2 unknowns. Several solution schemes are now possible. Here, we resort to template-based iterative methods since they are well-suited to threads. Computation on each mesh intersection point is governed by a thread. Even with iteration, several solution schemes are possible, each with different convergence characteristics. With Successive Over-Relaxation (SOR), a thread handling mesh point (i, j) must compute

$$\phi_{i,j}^{(n+1)} = (1 - \omega)\phi_{i,j}^{(n)} + \frac{1}{4}\omega[\phi_{i+1,j}^{(n)} + \phi_{i-1,j}^{(n+1)} + \phi_{i,j+1}^{(n)} + \phi_{i,j-1}^{(n+1)}]$$

where $\phi_{i,j}^{(n)}$ is the n -th iterate approximating the distribution ϕ at mesh point (i, j) , and ω is a positive constant called the acceleration factor.

The sequential implementation is straightforward. All data is local, and computations proceed in orderly fashion along rows or columns, without synchronization. The distributed implementation, however, requires synchronization. In practice, a red-black ordering of computations enables valid updates. Two passes are made over a mesh whose rows (or columns) are alternately colored red and black. On the first pass, red points are updated using black neighbors. On the second pass, black points are updated using red neighbors. The iteration terminates upon converging, i.e., when $|\phi_{i,j}^{(n+1)} - \phi_{i,j}^{(n)}| < \epsilon$, for all (i, j) and small, positive ϵ .

As mentioned earlier, threads are ideal for implementing iterations, and the red-black scheme is no exception. Pseudo-code for Ariadne threads is shown in Figure 15. By distributing vertical slices (sets of rows) of a mesh across a set of distributed processors, threads governing mesh point computations are also distributed. Each distributed process manages mesh points within its slice. With a single thread handling each mesh point, there will be as many threads as mesh points on the mesh.

The order in which threads are created is critical, since this can affect the order in which threads execute. Because Ariadne's internal scheduler uses a FIFO rule, threads within a priority class execute in order of creation. After each pass over the entire distributed mesh, processors synchronize to exchange boundary information. Two special threads are used for this purpose. One handles synchronization at the end of a 'red pass', and the other does the same at the end of a 'black pass'.

Several implementation aspects are worth noting. All computation is neatly packaged within a single `psor()` function which, of course, runs as a thread. This is both natural and intuitive. The user is spared the pain of writing code for organizing computation and synchronization. Function `main()` performs distributed system initialization using `a_dinit()`, and threads system initialization using `ariadne()`. Next, `main()` creates computation and synchronization threads and then hands the rest

```

struct sem* sync;           /* synchronization semaphore */
int done = 0;               /* termination flag */

void psor(int i, int j)     /* computation at grid point */
{
    oldV = infinity;
    do {
        compute a(i, j) based on red-black sor rule;
        a_waits(sync);      /* wait for synchronization */
    } while ((oldV - a(i,j)) > tolerance);
}

void redpass_sync()         /* synchronization after red pass */
{
    while (!done) {
        send leftmost column to left neighbor;
        send rightmost column to right neighbor;
        receive from right neighbor;
        receive from left neighbor;
        a_waits(sync);      /* wait for next iteration */
    }
}

void blackpass_sync()       /* synchronization after black pass */
{
    while (!done) {
        if (myself or any neighbor is still computing) {
            send leftmost column to left neighbor;
            send rightmost column to right neighbor;
            receive from right neighbor;
            receive from left neighbor;

            a_signalalls(sync); /* signal all threads waiting at the semaphore */
        }
        else {
            done = 1;
            a_signalalls(sync);
        }
    }
}

```

Figure 15: Distributed Red-Black SOR algorithm

Figure 16: Performance of SOR on different meshes

of the job over to the threads system. Creation occurs in the order: red threads, red-pass synchronizer, black threads, black-pass synchronizer. Because internal thread scheduling is FIFO by default, threads execute in order of creation.

Computation proceeds in time-steps, with all threads in a process waiting on a semaphore (`sync`) at the end of a pass. The red and black synchronizers signal all waiting threads at the end of their respective passes, so that another iterative pass can begin. Initially, each process starts out with the same number of threads, given that that mesh slices are equal-sized. Hence, initial processor loading is balanced. As execution proceeds, however, threads with iterates that converge leave the system and thus create load imbalance. Because threads represent static mesh locations, thread migration is unnecessary. Instead, processors exchange boundary data at the end of each pass. Barrier synchronization is inherent to the algorithm. Iterative schemes exist (e.g., *chaotic* relaxation schemes) which are synchronization free.

Performance

The SOR application was run on a SPARCstation 5 (Ethernet) LAN, with each machine hosting 32MB of real memory. In this case, Ariadne's messaging support came from the Conch [41] distributed system, which allows processors to be configured in a ring. The uniprocessor version of the program is also threads-based, but does not incur distributed system overheads. The environment was not dedicated to the application, though experiments were conducted during off-peak hours. Several mesh configurations were tested: 64×64 , 128×128 , and 256×256 , with $\epsilon = 5 \times 10^{-5}$. Mesh boundary values were set to $u(x, 0) = 0$, $u(0, y) = 0$, $u(x, 0.5) = 200x$, and $u(0.5, y) = 200y$, where $u(x, y)$ is the steady state heat value at (x, y) .

With one thread for each mesh point, large meshes generate large demands on memory, particularly when the number of processors is fixed. Unlike the previous examples where threads work awhile, create new threads and then die, each thread governing a mesh point stays around until its iterate converges. To avoid potential stack overflow arising during Conch's interrupt processing, each thread was given a stack of size 4KB. Though much larger than necessary for a thread's iteration work, these were sufficient to accommodate interrupt handlers invoked by Conch for message routing during thread execution. Arriving at memory limitations because of this, we resorted to using Ariadne's *common stack* feature: all threads run on a large, shared stack. This immediately reduced stack requirements to 1KB per thread. But because each thread's essential stack now had to be restored and stored, before and after its execution, respectively, execution cost went up.

Execution times for runs with the *common stack* and 4KB stacks are shown in Figure 16. Certain meshes were too large for small processor sets. For example, a 256×256 mesh with threads using 4KB stacks could not be run on fewer than 16, and a 128×128 mesh could not be run with fewer than 4 processors. A common stack made it possible to run a 256×256 mesh on 8 processors. But the run time ratio of the common stack version to the 4KB stack version came out to be 1.84, clearly indicating copy-related overheads. Common stack versions ran from 1.25 to 2.75 times slower than corresponding 4KB stack versions. Runs using swap space are not recommended because threads executing across iterations may generate severe page-faulting.

A simple and viable solution is to run larger examples in available memory, with a single thread governing iteration at multiple mesh-points. While this marginally increases the complexity of thread code, performance improves rapidly with increasing mesh sizes.

Granularity Effects

Computation granularity plays a key role in performance. To examine its effects, we repeatedly ran a 128×128 mesh, varying thread workload across runs; workloads ranged from 1 to 64 mesh points. With a granularity of 64 points per thread, iteration on each column of the grid was managed by two threads, one for updating red and the other for updating black points. The resulting effects on run time can be seen in Figure 17. For a fixed number of processors, a granularity increase implies smaller run time, with diminishing returns.

In this experiment, run time is unaffected by granularity for a 16-processor setup. Work generated by a 128×128 mesh is too small to justify the relatively high cost of communication and synchronization on 16 processors. Further, with more processes there is potential for higher load imbalance, as threads with converging iterates exit the computation in unpredictable patterns.

6.4 Particle Dynamics

Studies of particle dynamics are of considerable interest in materials engineering, fluid flow in porous media and electrical conduction. These phenomena are sometimes modeled as *random walks* on disordered clusters. The model described below is known as the *ant in the labyrinth* and is attributed to deGennes [28]. Briefly, a *random walker* (ant) moves randomly over certain accessible sites of a given

Figure 17: Effects of granularity in distributed SOR

```

void sim_part(int x, int y, int id) /* (x, y) is the particle's position,
                                     id is its unique identifier */
{
    int step;
    int source;
    int was_migrated;

    for (step = 0; step < MAX_STEPS; step++) {
        get new position (xN, yN) depending on the application;
        if ( (xN, yN) is not on myProcid) {
            source = my process id;
            a_migrate(process id of process hosting (xN, yN));
            was_migrated = TRUE;
        }
        else
            was_migrated = FALSE;
        if ((xN, yN) is not available) {
            if (was_migrated)
                a_migrate(source); /* migrate back */
            return;
        }
        if (was_migrated)
            send ACK to sender;
        x = xN, y = yN;
    }
}

```

Figure 18: Thread code simulating a Particle

lattice, where the *fraction* of accessible sites is q , $0 < q < 1$. We simulate the movement of the walker for T time-steps, and finally compute his mean square displacement (msd). The goal is to (empirically) determine msd as a function of q and T . Details on the model can be found in [28] and [24].

In this example, both thread migration and data transfer are used. The uniprocessor version of the application is simple to code. A 2-D grid is created to represent the lattice, and a set of sites is marked as inaccessible. A sequence of time-steps ensues, with each particle given a chance to move in a time-step. Random numbers are used to decide direction of movement. A particle moves to a chosen, neighboring grid site only if the site is accessible; otherwise, the particle stays put for the time-step. All information a processor requires is local.

While porting the sequential code to a shared or distributed memory environment is not difficult, some care is required. As in the previous example, grid slices may be allocated to distinct processors. Naturally, a processor hosting a slice must also host its walkers. Processes initialize their slices independently, marking certain sites as inaccessible and generating walkers. A processor need only be aware of the status of the slice it has been assigned. Since events occurring in a time-step may involve walkers located at other processors, and walkers may cross slice boundaries, inter-processor synchronization is required following each time step. At this time walkers may migrate from one process to another (between slices, or from one boundary to another).

A walker who migrates only to find his target site already occupied must migrate back to his original process and position. If his original position became occupied by another walker, in the meantime, a cascade of corrections may occur. For simplicity, we do not implement cascading retractions. Pseudocode for a walker is shown in Figure 18. Synchronization is based on a decentralized protocol, with each processor synchronizing only with its neighbors [24]. Though the application is regular and well-suited to distribution, load imbalance is a certainty. Initially balanced process loads become unbalanced as walkers begin to migrate across slice boundaries. Redistributing grid slices to balance load is possible, but is not pursued here. Work done by a process during a step is only a function of the number of walkers it hosts. This number varies dynamically, and is not easily predictable.

Each walker was implemented as a thread. Walker migrations across slice boundaries correspond to thread migrations between neighboring processors. The sequential threads version does not contain code to distribute grid slices or synchronize. Experiments were run on a SPARCstation 5 (Ethernet) LAN. One machine housed a master process (id 0) which controlled execution of up to 16 slave processes, each running on a distinct workstation. Each workstation hosted a single process related to this application. Lattices used were of sizes 512×512 and 1024×1024 .

Runs were made with $p = 1000, 2000$ and 4000 walkers, for a total of 100 time-steps. As before, all measurements exclude system initialization time. We report completion time, measured by the time elapsed between the first and last steps. As the number of processors was increased, processor workload per time step decreased.

As can be seen in Figure 19, substantial speedup may be obtained with many walkers. Increasing grid size (G) does not appear to impact execution time when the number of walkers (P) remains fixed. With 2000 walkers, a sequential version developed using threads ran faster than a 2-processor threads version. Moreover, this version even ran faster than an 8-processor threads version, for 1000 walkers. With 4000 walkers, we were forced to run the system with at least four processors because of memory

Figure 19: Performance of particle-dynamics problem

constraints.

7 Related Work

Ariadne is a novel, versatile and portable user-space threads system. Examples² and limitations of user-space implementations can be found in [25]. As evidenced by the growing number of operating systems promoting kernel-space threads (e.g., Solaris, Mach), efficient coupling of cheap user-space threads with kernel-level support can enhance development of concurrent applications [21, 2]. Because kernel-space thread implementations are OS specific, and tied to particular hardware, application portability is a serious concern. A possible solution is the use of the POSIX standard as a threads programming interface. It is left to be seen, however, whether particular implementations of the standard can circumvent operational and performance problems.

Threads have been proposed as basic units for enabling concurrency in many systems. For example, on shared memory multiprocessors, Ariadne multiplexes user threads on top of multiple processes. The PRESTO [4] system and μ System [6] also use processes in this manner, though the implementations are less general and do not support distributed memory architectures. Examples of other systems providing threads or threads-like interfaces with distributed memory support include IVY [20], Amber [9], Clouds [11], and distributed Filaments [15]. In IVY, a network-wide virtual address space, shared among loosely coupled processors, permits computations to access remote objects. The shared virtual

²For examples of threads systems see FastThreads [3], C Threads [39], Solaris Threads [31], Filaments [13], μ System [6], Pthreads [27].

memory allows threads and objects to migrate easily, since all processors have access to the same virtual address space. But in IVY, this requires maintaining coherence of copies of data resident on distinct processors. Such maintenance overheads and contention for multiple access to data degrades performance.

In the Amber system, though threads and objects may migrate, they are required to occupy the same address space on all processors through static preallocation. With a large number of threads, an Amber application may run out of address-space, regardless of the number of machines used. In contrast, an Ariadne thread occupies space only on the processor it resides on, and this space is returned to the system when a thread migrates. In Amber, a thread which accesses a remote object is moved to the object's host via a remote procedure call. Thus, repeated access to a given remote object may lead to thrashing.

The Clouds system offers a distributed environment made up of compute servers, data servers, and user workstations. It is based on the object/thread model. Objects are persistent and shared among threads; the runtime mapping between threads and objects is defined by invocations which are similar to RPC calls and can be nested. Objects are stored and shared through the Distributed Shared Memory (DSM) mechanism.

The Distributed Filaments (DF) system offers stateless threads, and is geared towards fine grained applications. Threads without state offer low-cost creation and context-switching, and appear to be ideal for fine-grained applications. The DF system is also based on the implementation of DSM. Like the Clouds system, it requires page consistency protocols. Stateless threads cannot be preempted or suspended before their execution is complete, and are hence not suitable for general applications such as in process-based simulation. Although Ariadne does not readily provide good performance in very fine-grained parallel applications (e.g., adaptive quadrature), increase in granularity almost always leads to improved performance.

In general, 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 impractical, 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 [34, 35]. Dynamic migration is usually addressed in the context of distributed operating systems, for example V [40] and DEMOS/MP [32].

Another form of thread migration, called *computation migration* [18], has been proposed as part of the Prelude language. This technique exploits compile-time transformations to migrate only the topmost frame of a stack. This frame may move from one node to another, repeatedly accessing objects, but must finally return to its source. Thus each migration requires a return. Computation migration is useful when thread stacks are large, enabling low migration costs because only necessary stack is migrated. A similar method is used for computation migration in Olden [8]. Here, the idea is to exploit compiler-based techniques to parallelize recursive C programs. During compilation, a remote access is resolved either by computation migration or page-level software caching. A combination of programmer hints, data-flow analysis, and heuristics are used in this resolution.

8 Summary and Future Work

Ariadne has proven to be an effective and portable threads system, ideal for programming in shared and distributed memory environments. It currently runs on the SPARC (SunOS 4.x, SunOS 5.x), Sequent Symmetry, Intel Paragon, Silicon Graphics IRIX, and IBM RS/6000 environments. The threads interface: basic, shared, distributed and customization primitives, provides the necessary functionality for threads-based parallel programming, and encourages tailored solutions to specific problems via thread-scheduling. Thread migration is a powerful feature, enabling a natural and direct representation of computational entities in many applications. It offers a facility for reduced-effort programming, since programs with thread-migration are similar to shared memory programs. Once threads migrate, all variable accesses become local.

Our studies indicate good performance on shared memory multiprocessors. Performance on distributed memory multiprocessors is largely a function of the algorithm and application, but our experience here has also been reasonably satisfactory. Of course, performance improves with coarser granularities, as is expected with distributed (workstation cluster) processing. We are currently binding Ariadne threads to kernel threads wherever these are efficiently supported, such as in the Solaris OS. This will enable cheap multiprocessing from within a single process, instead of via Ariadne's current multiple process scheme. This will also simplify inter-process communication in hybrid (combined shared/distributed memory) environments, and improve concurrency, since threads blocking on system services can generate kernel reschedules.

Combined with an effective object locator service, thread migration offers an ideal facility for adaptive workloads and reduced load imbalance. Domain redistribution can be effected by threads which know load, and the object locator can migrate executing threads to the new data locations. We intend to examine thread migration for load balancing in more detail.

Finally, Ariadne is currently being used to support two related environments. In the ParaSol parallel process-oriented simulation environment [22], threads implement simulation processes and operate in conjunction with optimistic and adaptive simulation protocols. In the investigation of efficient user-space implementations of network protocols, threads offer support for dynamic adaptations to network loads and active messaging [16].

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] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.
- [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] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [6] P. A. Buhr and R. A. Strooboscher. The μ System: Providing light-weight concurrency on shared-memory multiprocessor computers running unix. *Software-Practice and Experience*, 20(9):929–964, September 1990.
- [7] R. Butler and E. Lusk. User's Guide to the P4 programming System. Technical Report ANL-92/17, Argonne National Laboratory, 1992.
- [8] M. Carlisle and A. Rogers. Software caching and computation migration in olden. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, 1995.
- [9] 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.
- [10] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*, 1990.
- [11] 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.
- [12] D. L. Eager and J. Zahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Transactions on Computer Systems*, 11(1):1–32, February 1993.
- [13] D. R. Engler, G. R. Andrews, and D. K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. Technical Report TR 93-13a, Department of Computer Science, The University of Arizona, 1993.
- [14] A. Ferrari and V. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. In *Proceedings of the 4-th High Performance Distributed Computing Symposium, Washington, D.C.*, pages 211–218, August 1985.
- [15] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. Technical Report TR 94-11a, Department of Computer Science, The University of Arizona, 1994.

- [16] J. C. Gomez, V. Rego, and V. Sunderam. Tailoring Receive Threads to Suit Network Loads: Techniques and Experiments. Technical report, Department of Computer Sciences, Purdue University, December 1995.
- [17] C. A. R. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1):39–45, January 1971.
- [18] W. C. Hsieh, P. Wang, and W. E. Wehl. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 239–248, 1993.
- [19] J. M. Lemme and J. H. Rice. Adaptive quadrature algorithms for the ILLIAC IV. *International Journal of Computer and Information Sciences*, 9(1):63–72, February 1980.
- [20] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing*, pages 94–101, 1988.
- [21] 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.
- [22] E. Mascarenhas, F. Knop, and V. Rego. ParaSol: A Multi-threaded System for Parallel Simulation Based on Mobile Threads. In *Proceedings of the Winter Simulation Conference*, pages 690–697, 1995.
- [23] E. Mascarenhas and V. Rego. Ariadne User Manual. Technical Report 94–081, Computer Sciences Department, Purdue University, 1994.
- [24] E. Mascarenhas and V. Rego. DISplay: A Visualization and User Interaction Interface for Parallel Simulation Environments. *Computer & Graphics*, 19(5):739–753, 1995.
- [25] E. Mascarenhas and V. Rego. Ariadne: Architecture of a portable threads system supporting thread migration. *Software-Practice and Experience*, 26(3):327–356, March 1996.
- [26] E. Mohr, D. A. Kranz, and R. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
- [27] F. Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX*, pages 29–41, January 1993.
- [28] H. Nakanishi. Anomalous diffusion in disordered clusters. In P. J. Reynolds, editor, *On clusters and Clustering, From Atoms to Fractals*, chapter 27, pages 373–382. Elsevier Science Publishers B.V., 1993.
- [29] H. Nakanishi, V. Rego, and V. Sunderam. On the Effectiveness of Superconcurrent Computations on Heterogeneous Networks. *Journal of Parallel & Distributed Computing*, 24(2):177–190, March 1995.

- [30] J. M. Ortega and R. G. Voigt. *Solution of Partial Differential Equations on Vector and Parallel Computers*. SIAM, 1985.
- [31] 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.
- [32] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 110–119, October 1983.
- [33] J. Sang, E. Mascarenhas, and V. Rego. Mobile-Process-Based Parallel Simulation. *Journal of Parallel and Distributed Computing*, 33(1):12–23, February 1996.
- [34] 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.
- [35] J. M. Smith. A Survey of Process Migration Mechanisms. *ACM Operating System Review*, pages 28–40, July 1988.
- [36] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of the 1992 USENIX Summer Conference*, pages 1–9. SunSoft, Inc., 1992.
- [37] V. Sunderam, G. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences and Trends. *Journal of Parallel & Distributed Computing*, 20(4):531–546, March 1994.
- [38] A. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.
- [39] 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.
- [40] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12, 1985.
- [41] B. Topol. Conch: Second generation heterogeneous computing. Technical report, Department of Math and Computer Science, Emory University, 1992.
- [42] L. Turcotte. A Survey of Software Environments for Exploiting Networked Computing Resources. Technical report, Engineering Research Center for Computational Field Simulations, Mississippi State, January 1993.
- [43] T. von Eicken, D. Culler, S. Goldstein, and K. Schauer. Active messages: A mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, pages 256–266, May 1992.