INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

A Performance Evaluation of Fine-Grain Thread Migration with Active Threads

Boris Weissman, Benedict Gomes, Jürgen W. Quittek,
Michael Holtkamp[1]

TR-97-054
December 1997

1. Technical University of Hamburg-Harburg, Germany

Abstract

Thread migration is established as a mechanism for achieving dynamic load sharing and data locality. However, migration has not been used with fine-grained parallelism due to the relatively high overheads associated with thread and messaging packages. This paper describes a high performance thread migration system for fine-grained parallelism, implemented with user level threads and user level messages. The thread system supports an extensible event mechanism which permits an efficient interface between the thread and messaging systems without compromising the modularity of either. Migration is supported by user level primitives; applications may implement different migration policies on top of the migration interface provided. The system is portable and can be used directly by application and library writers or serve as a compilation target for parallel programming languages. Detailed performance metrics are presented to evaluate the system. The system runs on a cluster of SMPs and the performance obtained is orders of magnitude better than other reported measurements[2].

2. A shorter version of this report is accepted by IPPS/SPDP98

1 INTRODUCTION

In recent years, the multi-threaded programming model has grown increasingly popular, propelled in part by the increasing availability of shared memory multi-processors (SMPs). Programming languages make use of the threaded model both for expressiveness (particularly in graphical user interfaces) and to exploit the parallelism available on an SMP. The convenience of the threaded model arises from the ease of sharing data between threads, and from the availability of automatic load balancing since threads, once blocked, may continue to execute on any available processor.

In a hybrid distributed system, where the nodes consist of SMPs, it becomes necessary to either provide a mixture of programming styles or to extend a traditional distributed or shared memory programming style. Given these choices, extending the powerful and relatively simple (compared to message passing) threaded programming model to a distributed system is appealing. To fully support the threaded programming model in a distributed environment it is necessary to support (a) a from of distributed shared memory to permit the sharing of data between non-local threads and (b) thread migration to achieve the relatively transparent load balancing available on shared memory systems. A limited form of computation migration is to permit thread creation on remote clusters. However, migration after creation may be necessary in order to achieve load balance while retaining the simplicity of the natural threaded model, particularly with long running threads.

Why not threads? Poor Performance!

A major problem with using the threaded model for fine-grained concurrency is performance. While most modern thread packages are adequate for medium grained concurrency, the overhead costs are unacceptable for fine-grained parallelism. For instance, thread creation using Solaris threads on our SPARCstation 10 system takes 1 to 10 milliseconds; the median thread lifetimes for our application are of the same order. The performance of thread migration, which is necessary for load balance in the threaded model, is also affected by the thread system overheads. In an effort to avoid these overheads, while maintaining some of the benefits of threads, many researchers have turned to simpler, non-blocking threadlets (Filaments [LFA96], Cilk [BJK+96], Multipol [WCD+95]).

Why not threadlets?

Though non-blocking threadlets do provide high performance, they are not adequate in their expressiveness for most modern explicitly parallel object-oriented languages (including Java, Ada, CC++ and locally designed language, pSather [SO96] [QW97]) which make use of blocking threads. Using non-blocking threadlets requires shifting to a radically different programming style, such as continuation passing or implicitly parallel functional programming.

Why is thread performance poor?

There are several reasons for the poor performance of commercial thread packages. Kernel level threads, as found in NT [ISW97] involve expensive kernel traps [Ber91]. Even the user level thread packages available commercially have high thread creation costs; they appear to be targeted at medium grained parallelism, particularly in applications such as user interfaces where expressiveness rather than performance is the central issue.

Why is migration performance poor?

In addition to the problems of commercial thread packages, the performance of thread migration depends on the communication system. Many migration systems make use of kernel traps for either the thread system [ISW97] or for the communication system [MR96] [NM97]. Performance also depends on the degree of integration between the thread and communication systems. Nexus [FKT+96] presents an interface to a combined thread and communication systems; however, it makes use of off-the-shelf thread and communication components. A third potential reason for the poor performance of thread migration is the pointer fixing after migration that is performed by systems such as Ariadne [MR96] - see Section 3.1.

Improving Performance

We demonstrate a user level thread package, Active Threads, that achieves high performance with low overheads for thread creation and other thread operations. Thread creation overheads are comparable to threadlets overheads. We also utilize user level messaging, with support for concurrent access to the network interface. A novel feature of the thread package is an extensible event mechanism that permits a close integration between the thread and message passing systems. The system provides primitives for thread migration, but does not dictate a particular thread migration policy. We demonstrate this flexibility by exploring the effect of different policies on several applications.

In summary, this paper makes the following contributions:

- A fine-grained thread migration system with user level threads and messaging. High performance is achieved on a cluster of symmetric multiprocessors using a low-cost commodity network (Myrinet).
- An extensible thread event mechanism that permits the flexible and efficient integration of independent thread and communication systems.
- Detailed performance metrics for thread migration, and the ability to compare the effects of different migration policies on application performance. Since the thread migration system is completely at the user level, operating system dependencies are eliminated, permitting a better comparison of underlying costs.

An underlying shared address space using global pointers is assumed. We provide two different examples of such systems; others are possible and the thread package is not tied to any particular implementation.

An implementation of thread migration involves four basic components, each of which involves design decisions

- The transfer of thread local data. (Section 3.1)
- The access of global data. (Section 3.2)
- The thread system (Section 4).
- The messaging system (Section 5).

After describing the system components, we describe micro-benchmarks that measure basic thread performance and compare these results with existing systems. We then go on to describe the use of thread migration in a multi-threaded version of the ‘grep’ utility. We demonstrate the impact of several different simple thread migration policies; in particular we consider the significant improvement in performance that may be achieved by using mi-

gration to improve data locality. We also describe similar results on a function integration application.

2 THE MIGRATION INTERFACE

Thread migration is triggered by one of the following calls into the thread package. The return value of each call is a code indicating success or failure.

```
int steal(bundle_t b);
   Steal a thread from any other node.

int steal_from(bundle_t b, node_t from);
   Steal a thread from the node 'from'.

int push(bundle_t b, node_t to);
   Push any other blocked thread to the node 'to'

int push_self(bundle_t b, node_t to);
   Push the current thread to the node 'to'
```

The first argument of each call refers to the thread bundle (each thread is associated with a thread bundle, described in more detail in Section 4). The interface allows either pushing work to or stealing (i.e. pulling) work from other nodes. The general calls, steal and push, have variants that permit the more exact specification of which thread to migrate. In a push, a thread may elect to either push itself or to push any available (i.e. runnable) thread to the destination node. The steal calls may elect to either steal any available work or to steal data from a particular other node. In addition to the above calls, which are synchronous, asynchronous versions are also provided.

3 TRANSFERRING STATE

In order to effectively move a thread to a different processor, it is necessary for the transferred thread to correctly access all related data and resources. All data that is local to the thread (the stack and any thread-local heap) may be copied to the destination processor. However, a problem may arise, since the addresses on the target processor may be different from the original addresses and internal pointers may no longer be valid. Furthermore, data that is not globally replicated or transferred along with the thread will not exist on the destination node (see Figure 1).

3.1 Thread Local Data

Two approaches may be used to handle local pointers when threads are migrated. In the first approach, local pointer values may change after the thread has been migrated, and local pointers must be updated after migration. The second approach is to partition the address space such that local pointers may maintain the same values after migration.

In order to update pointer values after migration it is necessary to register stack pointers. If registration is left to the programmer, pernicious bugs can arise due to errors in registration. A more basic issue is that it is frequently impossible to identify and correctly update pointers that have been stored in registers, unless a great deal of compiler and language level support is provided. The resulting system is consequently limited to a particular

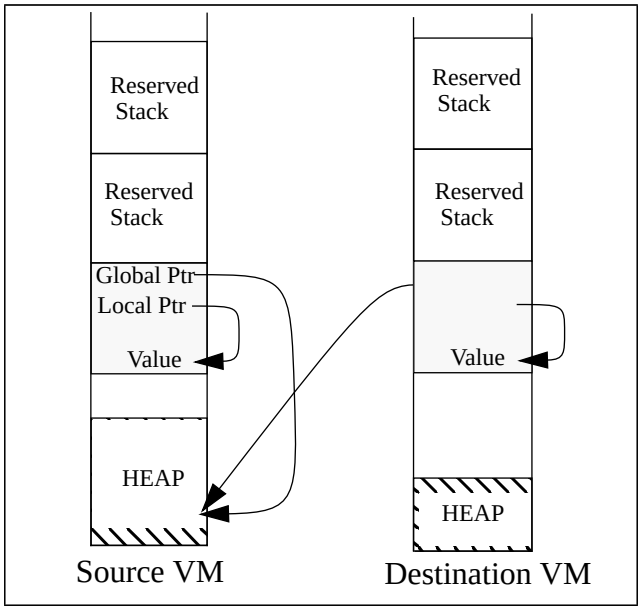

Figure 1: Transferring thread data

language and compiler system. Ariadne [MR96] requires user registration but provides no solution to handling registers as pointed out in [ISW97]. Emerald [JLHB88] is a full fledged system, with compiler and even kernel level support. Even with the necessary compiler support, updating pointers will exact a certain run-time penalty over and above the cost of moving the thread data.

Since we were interested in a portable, high performance system, we chose the alternate approach of storing the thread local data at the same location in the address space on the destination node, so that no local pointers need be updated. A pre-defined area of the virtual memory space is reserved for each thread stack on all nodes. This is essentially the same approach as used by Millipede [ISW97], Amber [CAL+89] and UPVM [CKO+94]. Indeed, Amber as a successor system to Emerald, chose this approach for similar reasons [CAL+89]. Using this approach, the total number of threads is limited by the address space available on a single node and does not scale with the number of nodes. However, with the increasing availability of processors with 64 bit address spaces, or on smaller sized clusters, this limitation is not a serious issue. Note that the limitation is the virtual address space of a single processor, not the physical memory present, as claimed in [CHM97].

We considered, but have not yet implemented, two possible improvements to this scheme that would reduce the memory requirements. Instead of mapping a large thread stack area, it is possible to - incrementally map memory segments, as needed. A probabilistic approach that avoids pre-allocating memory addresses would be to only permit migration onto those clusters where the necessary portion of the address space is currently available

3.2 Shared Data

In addition to thread local data i.e. the stack and the local heap, a thread may also access data that is shared between multiple threads. Examples of such shared data include resources located on a particular machine and global synchronization objects.

After a thread has migrated, if pointers to this shared state are to still be valid, it is nec-

essary for the pointers to indicate the nodes on which the data resides. A simple approach is to adopt the same partitioning of the global address space mentioned in the previous section. References to non-local data will result in a trap, which can either forward the reference to the proper location or transfer the required data.

Another approach is to use global pointers, which create the illusion of a shared address space. Global pointers, which are commonly used in parallel languages, usually encode the node location in the unused bits of a standard pointer. Global pointers incur a certain overhead whenever they are dereferenced, even when the data is locally available since the location of the data must be ascertained before dereferencing. Hence, it is desirable to avoid the use of global pointers where possible such as when accessing thread local data, as described in the previous section.

In the parallel language pSather (which uses Active Threads as a compilation target), the node location is encoded in the unused high bits of the address [Fle97]. In the C++ based library [Hol97] we implemented a set of classes encapsulating global pointers. They hide access to global pointers by pointer overloading and offer a shared variable programming model for designated data types.

Though the thread migration system described here requires a shared address space, it does not depend on any particular implementation.

4 THE THREAD SYSTEM

Active Threads is a general-purpose portable and extensible light-weight thread package for uni- and multiprocessors targeted at irregular applications. Active Threads is capable of handling millions of threads on a variety of hardware platforms. Unlike most other thread packages, which utilize hard-coded scheduling policies, Active Threads provides a general mechanism for building data structure specific thread schedulers and for composing multiple scheduling policies within a single application. This allows modules developed separately to retain their scheduling policies when used together in a single application. Flexible scheduling policies can exploit the temporal and spatial locality inherent in many applications. The extensibility and flexibility of the scheduling mechanism enables easy integration of the thread system with other systems such as high-performance network interfaces and garbage collection. Such interoperability is not commonly supported by general-purpose thread packages.

The details of the Active Threads architecture, and API are given in [Wei97]. In this section, we concentrate on the aspects of the Active Threads organization that enable us to integrate threads with communications, efficiently implement thread migration primitives and to freely experiment with different migration policies.

4.1 Threads and bundles

Active Threads supports general-purpose blocking threads. *Threads* are units of (potentially parallel) execution that share the address space and other system resources. Each thread maintains a set of registers including a program counter, a stack that stores the thread's local variables and a small thread heap also called *thread local storage* (TLS).

Groups of logically related threads with common properties are organized into *thread bundles*, or simply *bundles*. All threads in a bundle share the same scheduling policy. Scheduling policies for different thread bundles can be completely independent from each

other. A single application may create thread bundles with different scheduling policies such as FIFO, LIFO, priority, processor affinity scheduling, etc. Moreover, a scheduling policy for a bundle is not fixed and can change dynamically. While bundles were mainly designed to enable compositional development of parallel software and encapsulate scheduling policies together with other module resources, bundles and the associated scheduling event mechanism have proved to be an excellent abstraction to enable fine-grained thread mobility in distributed environments.

Each thread's state information is encapsulated in the *thread context block (TCB)*. On context switch, the register state is stored at the top of the thread stack. A pointer to this area is kept the thread context block. The context block also keeps other miscellaneous thread information and a pointer to a thread bundle.

We now turn to the Active Threads scheduling mechanism.

4.2 Event Mechanism

Active Threads provides no hard-coded scheduling policy. Instead, it supports a general mechanism upon which different specialized scheduling policies can be built.

All thread scheduling decisions are made by the bundle to which the thread belongs. The bundle is free to maintain any scheduling data structures that fit most closely the semantics of the thread group.

Different susbsytems communicate with bundles by vectoring *scheduling events*. Bundles encapsulate all aspects of scheduling and must provide event handlers for all scheduling events. There are no restrictions on the implementation of such event handlers.

The relationship between bundles and other subsystems as well as the direction of the event flow are shown in Figure 2.

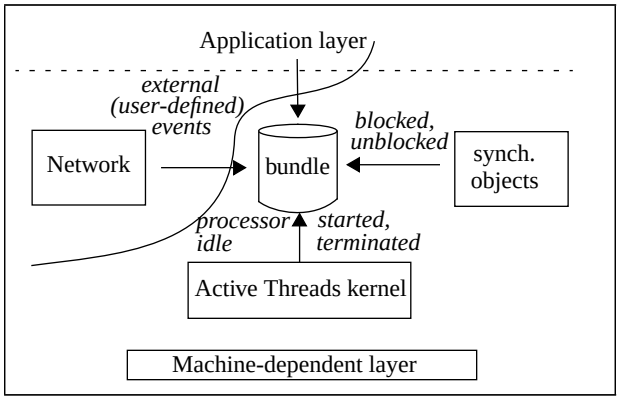

Figure 2: Scheduling events in Active Threads

All events are partitioned into two groups: internal Active Threads events and external (user-defined) events. Internal events deal with common thread operations: thread creation, termination, blocking, unblocking, dispatching by the processor, etc. All internal events, their logical origins and a short description of the information they provide to the thread bundle are given in Table 1[3].

thread created (kernel)	informs about creation of a new thread.

Table 1: Internal events

thread started (kernel)	informs about thread start-up; enables lazy stack allocation policies.
thread terminated (kernel)	informs about thread termination.
thread blocked (synch. object)	informs about thread blocking on a synchronization object
thread unblocked (synch.objects)	informs about thread unblocking.
bundle created (kernel)	informs about creation of a new child bundle.
bundle terminated (kernel)	informs about termination of a child bundle.
processor idle (kernel)	requests more threads for dispatching by the idle processor.

Table 1: Internal events

A bundle encapsulates all scheduling decisions for a group of logically related threads. The system is extensible: new scheduling policies are implemented by extending the bundle library. New synchronization objects can be added transparently to the rest of the system - synchronization objects may have arbitrary semantics; the event mechanism provides the interface for communication with the rest of the system. Active Threads are distributed with a variety of commonly used synchronization objects: spinning and blocking mutual exclusion locks, semaphores, reader/writer locks, and general condition variables. [Wei97]

Active Threads bundles can be combined into hierarchies and events can be forwarded between bundles. For instance, if a bundle that received a *processor idle* event does not have any runnable threads, it can forward the event to some other bundle or bundles. More details on the scheduling event mechanism and examples can be found in [Wei97].

A user can also define events logically originating outside Active Threads and register event handlers for these events. For instance, such events can be used to check the network for incoming messages whenever a processor becomes available or to perform a unit of garbage collection when all processors in the node are idle. This functionality can be also used to implement thread migration transparently to the rest of the thread system.

4.3 Scheduling Policies

An Active Threads bundle must implement event handlers for all eight internal events. No restrictions are placed on the bundle internals. The implementations are free to select the best data structures possible to implement the scheduler. For instance, if the number of threads is known statically (or even dynamically at bundle creation time), the scheduler may keep threads in an array to minimize rescheduling overhead. In simple dynamic cases, a linked list may be sufficient. However, to implement priority scheduling, more complex (and expensive) data structures such as priority queues are used. In general, the least expensive bundle that fully implements the desired functionality should be chosen. Active Threads are distributed with a library of bundles that support commonly used scheduling

3. "kernel" as used in the table means Active Threads kernel rather than OS kernel.

policies such as FIFO, LIFO, lazy stack allocation, etc. Some bundles in the library try to minimize cache misses and bus traffic by exploiting processor locality information for thread scheduling.

5 COMMUNICATIONS

Aside from context switching times, the other major overhead involved in thread migration arises from the system overhead of most messaging systems. Much of the overhead is caused by repeated copying of messages to and from buffers located in the OS kernel address space. The Active Message approach [vCGS92] eliminates much of the system overhead by handing control of the delivered message directly to a user-level message handler. Although the thread migration mechanism described in the following section does not rely exclusively on the communications style of Active Messages, user-level management of communications is indispensable for achieving low migration latencies and high throughputs. We have implemented a variant of the Berkeley Active Message communications system that supports the general Active Message communications style [vCGS92], but also extends the Active Message interface and functionality in several important ways.

Active message communication consists of matching request and reply operations, each of which invoke their respective handlers. In general, message handlers must be non-blocking functions that do not trigger any additional messages with one exception; request handlers may send at most one reply to the requesting node.

The active message system, as originally conceived and available in the current Berkeley distribution, was not designed to run on multi-threaded SMPs. Multi-threading raises the possibility of contention for the network device; consequently, all accesses to the messaging system must be properly protected.

Much of the overhead involved in a message transfer (i.e. the message packaging) is actually incurred on the host processor. Therefore, on an SMP it is possible to increase the message throughput by performing the message packaging in parallel on the processors of the SMP.

Another extension of Active Messages functionality deals with the kinds of operations allowed in the remote handlers. Active Message handlers are non-blocking and complex protocols must be implemented in applications to achieve the desired functionality. For instance, it is not straightforward to use primitive active messages to gain an access to a remote resource shared by several threads. Other researchers have pointed out similar problems with restricted handler functionality [FKT96].

Our communication system retains the Active Messages programming style and high performance. In addition, it provides the following features that enable closer integration of communications with threads and synchronization:

- Support for networks of symmetric multiprocessors; all communication operations can be executed concurrently. Since it is possible to parallelize the message processing that occurs off the network interface, the message throughput can actually be improved in concurrent contexts.
- Support for remote thread creation. As advocated by Nexus [FKT96], we support two kinds of remote handlers - threaded and non-threaded. Threaded handlers create a new remote thread and immediately send a message acknowledgment; the resulting thread is free to initiate other communications requests or bulk transfer operations, block, allocate

memory, etc. Due to the low thread creation overheads in Active Threads (section 7), the cost of threaded handlers is very close to that of non-threaded handlers. The non-threaded handlers are the original Active Message style handlers and incur no performance penalty.

The performance of the message system is strongly affected by the processor performance. On our system, which consists of the network of 50 MHz HyperSPARCs with 4 processors per node, a one way latency for a message with a payload of 5 words is around 17μs. A bulk transfer of 1K takes 560μs and is mostly constrained by the host processor and I/O bus speed rather than the network interface.

6 THREAD MIGRATION MECHANISM

We view thread migration as a natural extension of thread scheduling for a single (possibly multiprocessing) node to a distributed environment. Migration is implemented by defining new bundles that are extended with special user-defined migration events. This enables seamless integration of migration with other thread subsystems. Migration happens transparently to other thread activities and bundles and threads that do not support migration need not be concerned with any new and unforeseen effects. Different migration policies are implemented by defining new bundles and can coexist in a single application. Though the mechanism is general, we show (Section 7) that we pay a very small price for this generality. In fact, thread migration in Active Threads outperforms other systems with hard-coded migration policies by orders of magnitude in some cases.

To show the universality of the event-driven approach, we will consider an example of a thread relocating itself to a different node. To enable any existing bundle to support this functionality, only two new user-defined events need be added:

1. *outgoing* event; this event is generated by the Active Threads kernel on behalf of the user thread after the thread is safely stopped and its context is saved. The reference to the thread's TCB is forwarded to the bundle. The event is handled by the user-defined *outgoing* event handler which is responsible for all operations associated with the transfer of the thread's state over the network.
2. *incoming* event; generated externally to the thread system by the network message handler on message arrival. The incoming event handler is responsible for interpreting and unpacking the message, allocating a new TCB on the destination node, etc. Having finished these auxiliary operations, the event handler adds the arrived thread to the bundle's internal scheduling data structures.

Figure 3 captures the essence of the underlying mechanism.

At time 1, a user thread executing on node 1 calls *at_push_self(bundle, 2)*. The Active Threads kernel stops the calling thread and saves its register state in a predefined area (at the top of the thread's stack). It then dispatches an *outgoing* thread event on behalf of the stopped thread to the corresponding bundle.

At time 2, the *outgoing* event handler catches the event vectored by the kernel. As an optimization, the handler may pack all the pieces of the thread's state such as its TCB, thread-local storage (TLS), and the thread stack into a contiguous area (our implementation performs this optimization). The handler then initiates the network transfer by handing the

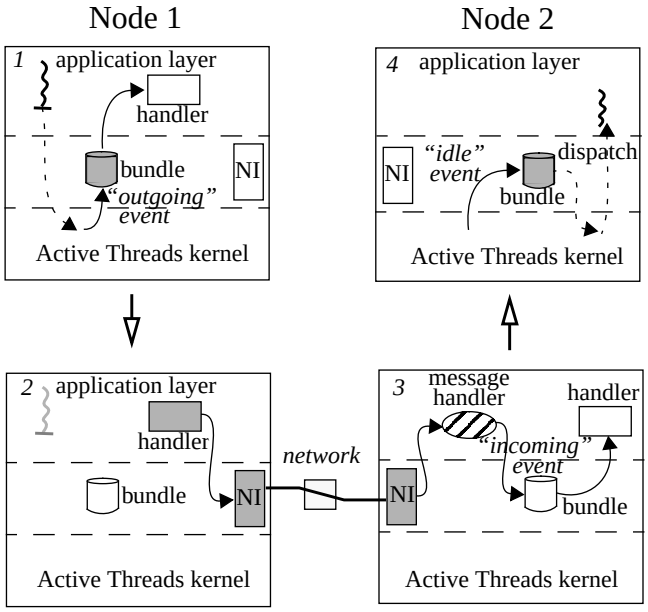

Figure 3: Events and thread migration.

message over to the network interface (NI).

At time 3, the incoming message handler of the destination node pulls the thread's state off the network and dispatches an *incoming* event to the destination bundle. The *incoming* event handler unpacks the message, obtains a new TCB from the thread system, initializes it with the proper values and adds a new thread to the bundle's internal scheduling data structures. The last step can be performed, for example, by dispatching a *thread unblocked* event for a newly arrived thread.

Finally, at time 4, the Active Threads kernel detects that one of the processors of node 2 becomes idle and dispatches a *processor idle* event. At this point, the newly arrived thread is indistinguishable from all other threads in the same bundle. It is handed over to the Active Threads kernel for dispatching on the idle processor.

The other primitives in the thread migration interface are handled analogously. For *push()*, the first step is simplified since the migrating thread is already stopped. Different versions of *steal()*, initiate migration by sending a message to the thread supplier node first. The subsequent steps are the same as for *push()*.

7 MICROBENCHMARKS

In the following section we present microbenchmarks that measure the performance of the thread migration system. We start by presenting the performance profile of the underlying thread system on a variety of modern platform. We then briefly discuss the organization of our NOW. We then present a detailed performance analysis of our thread migration system in terms or the following parameters:

- point-to-point latency
- initiating node overhead
- thread migration throughput

7.1 Threads and synchronization

We have measured the performance of Active Threads on a variety of hardware platforms: different models of SPARC symmetric multiprocessors, Intel Pentium Pro, DEC Alpha AXP, and HPPA 1.1. Performance measurements for Active Threads primitives on some of these platforms are shown in Table 2.

Operation	UltraSPARC-1, 167 Mhz	Intel PentiumPro, 200Mhz	DEC Alpha AXP 250Mhz	HPPA 9000/755, 99Mhz
thread create	1.3	1.4	1.0	2.0
null thread	5.6	4.4	2.9	7.2
context switch	1.7	1.5	1.1	3.0
uncontested mutex	0.4	0.5	0.3	1.0
uncontested sema.	0.4	0.5	0.3	1.0
mutex try	0.2	0.2	0.1	0.3
semaphore try	0.2	0.2	0.1	0.3
mutex ping-pong	6.0	3.4	2.9	7.9
sema. ping-pong	6.0	3.7	2.8	8.5

Table 2: Cost of thread and synchronization operations in Active Threads, μs.

The presented thread creation overhead includes thread stack allocation. Thread creation overhead with lazy stack allocation is somewhat smaller. In comparison, a null procedure call on the 167Mhz UltraSPARC-1 takes 0.75μs when register window overflow occurs and 0.08μs without window overflow.[4] Thus, thread creation overhead is almost as expensive as null function call with window overflow and only about an order of magnitude more expensive than a null call that does not cause a window overflow.

A brief description of the semantics of thread operations used in the benchmarks follows. The reported results are obtained by averaging over 1,000,000 executions of each operation.

- *thread create* - creation of a new thread, including stack allocation, but excluding context switch to the newly created thread.
- *null thread* - creation, execution, and termination of a thread whose body is a single null function.
- *Context switch* time is measured by having a thread yield execution to another thread.
- *Uncontested mutex* and *uncontested semaphore* - cost of mutex lock and semaphore wait operations in the absence of contention.
- Mutex and semaphore *try* - cost of possibly unsuccessful synchronization.
- Mutex and semaphore *ping-pong* operations repeatedly synchronize two threads with one another in a manner similar to that used to measure the synchronization cost of Solaris Threads in [PKBS+91]. Ping-pong timings include the cost of synchronizing two threads. Another related metric, per-thread synchronization overhead, as reported in

4. gcc v2.7.1, compiled with -O2

[PKBS+91], is exactly one half of the reported numbers.

A detailed comparison of the thread and synchronization performance of Active Threads and other modern research and commercial thread systems is given in [Wei97].

7.2 *Thread Migration*

Our main platform is a cluster of SPARCstation-10 workstations connected together by

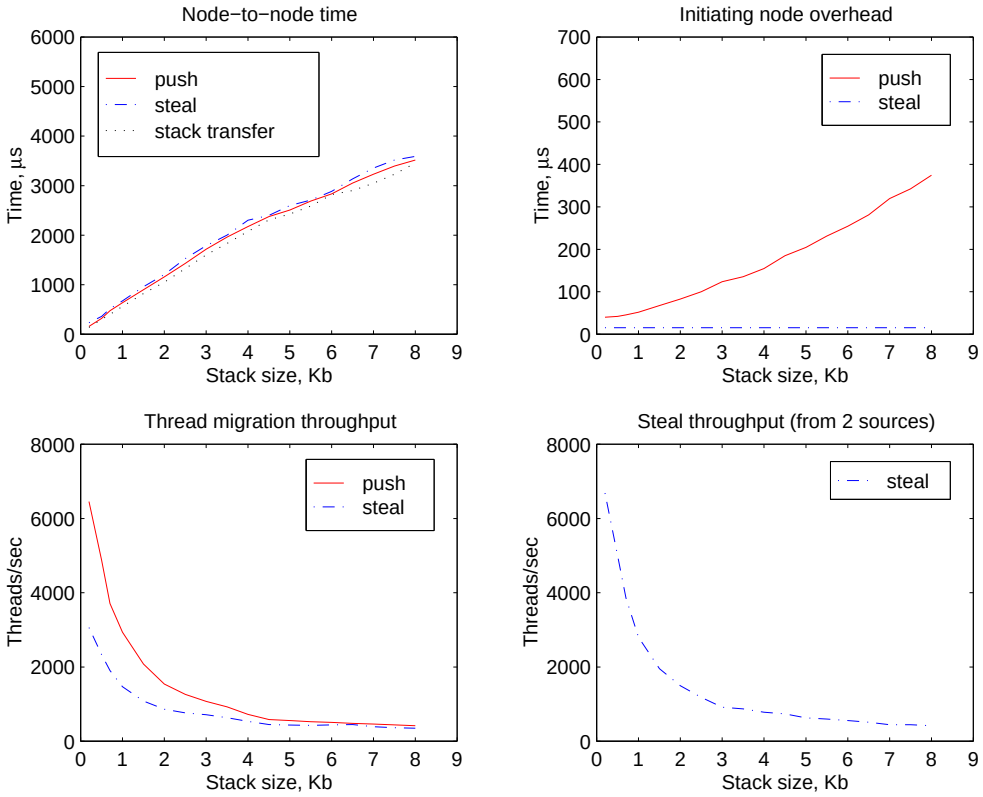

Figure 4: Migration latency and overhead in Active Threads.

Myrinet [BCF+95]. Each workstation has four 50 Mhz HyperSPARC processors. The Myrinet NI is an I/O card that plugs into the standard SBus. It contains a 32-bit RISC "LANai" network processor, DMA engines, and local memory (SRAM). The NI's memory is mapped into the user process address space and can be accessed through load/stores to mapped main memory addresses. The Myrinet network consists of crossbar switches with eight bidirectional ports that can be linked into arbitrary topologies. The network is constructed of low-cost off-the-shelf commercial hardware that is readily available for a variety of architectures. Figure 5 shows the configuration of each node of our network.

The costs of different thread migration operations for this network configuration are presented in Figure 4. All graphs reflect an average over 1,000 migrations.

Point-to-point time for a push operation includes blocking a thread on a source node, transferring its state to the destination and unblocking there to resume execution. It was

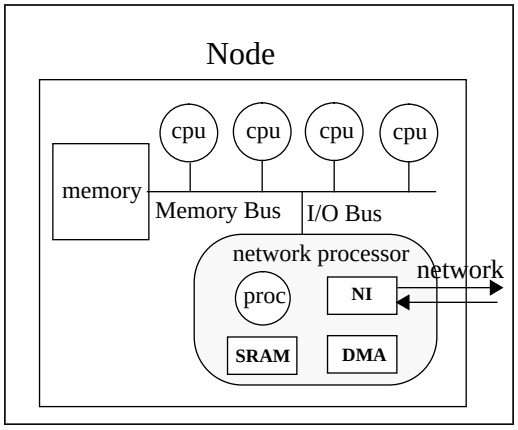

Figure 5: The structure of a processing node.

measured by having a thread repeatedly ping-pong between a pair of nodes. A similar measurement for a steal operation involves thread chains: a stolen thread initiates the next steal upon arrival after it has unblocked on the destination node. Both measurements present wall clock time elapsed from operation initiation to completion obtained with the help of a high-resolution nanosecond hardware timer. As the figure shows, both costs are heavily dominated by the transfer time of the active portion of the stack (the stack size is designated by the horizontal axis in all the graphs). The steal operation is slightly more expensive than the push operation - the destination node must send a message to the source (actually, it creates a very lightweight thread remotely) to set up and initiate the DMA of the source node network interface.

Migrating a null thread (a stack of 112 bytes for SPARC v8 and gcc 2.7.1) to a remote node takes 150μs, while pushing a thread with an active (used) stack of 1Kb takes 630μs.

While point-to-point times are important measures of thread migration performance, many load balancing strategies avoid paying the full price by initiating migration early while the processors are still busy. Thus, in the case of push, the cost of migration can be reduced to the fairly small overhead needed to initiate the transfer. The source node then resumes the computation while the thread transfer is still in progress. The push overhead is dominated by the copying of the thread state to SRAM of the network processor in order to enable the network DMA. This copy overhead is, in turn, determined by the speed of the I/O bus and grows linear as the active thread stack size increases.

The overhead for a thread steal consists of two parts - the steal initiation on the requesting node and the service overhead on the remote node. The former is fixed and fairly small - only 8μs on our system. The latter is exactly the thread push overhead.

On our system, thread migration overheads are at least an order of magnitude lower than full point-to-point latencies for medium size stacks. For instance, the push overhead for a null thread is 40μs and the overhead for a thread with an active stack size of 1Kb is only 52μs.

For applications with small tasks, but highly dynamic and unpredictable loads, frequent thread migration may be necessary and throughput rather than latency can be the key factor. The migration throughput may also affect the programming style as well as application design. For instance, a single producer/multiple consumer pattern is well known for its simplicity and may be adequate to capture the semantics of many applications. However, designating a single node as a producer imposes a bandwidth constraint on the system in

order to keep all the consumers busy.

From Figure 4, it is evident that Active Threads achieves a fairly high throughput, on the order of thousands of threads per second for small to medium thread stack sizes. Steal throughput is lower than push throughput, in particular for light-weight threads. This is due to the cost of the initiating message, the effect of which is amortized as the thread stack grows in size.

The last graph of Figure 4 demonstrates the scalability of our implementation. The throughput for small threads is essentially doubled when a node starts prefetching from two sources simultaneously.

7.3 Comparisons with other systems

It is hard to directly compare the performance of systems that support thread migration because of hardware differences such as processor and network speeds and the structure of memory hierarchy. Reported performance parameters also vary. Still, we find it instructive to list the performance characteristics of several recent thread migration systems, along with the hardware characteristics of each system.

System	null	1K	2K
Ariadne (SS5, Ethernet)		11ms	14ms
Millipede (Pentium/fast Ethernet)	2ms		?70ms?
PM^2 (ALPHA/1 processor)			2.2ms
ActiveThreads (SPARCstation-10/Myrinet)	40µs 150µs	52µs 630µs	68µs 1.1ms

Table 3: Migration time on various systems as function of thread stack size.

The numbers in Table 3 warrant some explanation. In terms of pure processing power, Active Threads run on a relatively modest distributed platform - all the other systems employ hardware with the clock speed at least several times faster than our 50Mhz SPARCstation-10s. Unlike previous systems, threads are migrated between symmetric multiprocessors with additional protection cost. Our preliminary experiments on a Sun Enterprise 5000 demonstrate latencies reduced by a factor of 3.

Millipede [ISW97] reports the “average latency” of thread migration to be 70ms; however, the average thread stack size is not specified. In our own experiments, the average active stack size of a migrating thread was between 1K and 2K.

The PM^2 [NM97] latencies are for thread migration between processes on a single 1 processor DEC ALPHA workstation using PVM for interprocess communications.

Both overhead and point-to-point latencies for Active Threads are presented in Table 3. In contrast to the other systems, Active Threads supports thread operations as well as communication at the user level. This allows us to keep operation overheads small, relative to the full operation latency. The overhead on other systems, although not reported, must account for most of the point-to-point latencies: Ariadne [MR96] traps to the OS kernel for TCP/IP communications; Millipede is based on kernel-level threads and must trap to the kernel for most thread operations; PM^2 traps to the kernel for interprocess communication. As a result, Active Threads latencies are at least an order of magnitude lower than those of the other systems while the migration overhead can be several orders of magnitude lower.

8 APPLICATION STUDIES

We have implemented several non-trivial threaded applications that rely on thread migration for load balancing as well as to improve locality. The aim of these experiments is to illustrate the performance of our thread migration system and its ability to utilize different migration policies rather then designing new migration heuristics.

8.1 at_grep

The first application is a threaded version of “grep”, a standard Unix utility for regular expression-based search. It is based on a version of grep by Ron Winacott[5]. “at_grep” supports the full functionality of “grep”. It also supports several additional features such as the recursive traversal of directories.

We start by describing the thread structure of at_grep and then examine how different migration policies can be used by this application.

The basic structure of at_grep is quite simple[6]: it parses the input regular expression and arguments and then starts a recursive search over the directory tree. Matches are printed as the search continues.

There are two basic types of threads: *search* threads and *cascade* threads. A search thread is created for each non-directory file found while descending the directory tree. Such a thread searches a single file and outputs the result. A cascade thread is created whenever the search comes across a new directory. Cascade threads are essentially producer threads - they call into the OS kernel to obtain the directory information structure and then create search threads for files in the directory and new cascade threads for subdirectories. If we view the thread creation tree, the interior nodes of the tree are cascade threads while the leaf nodes are search threads. Graphically, the relationship between directories, files, and different kinds of threads is shown in Figure 6.

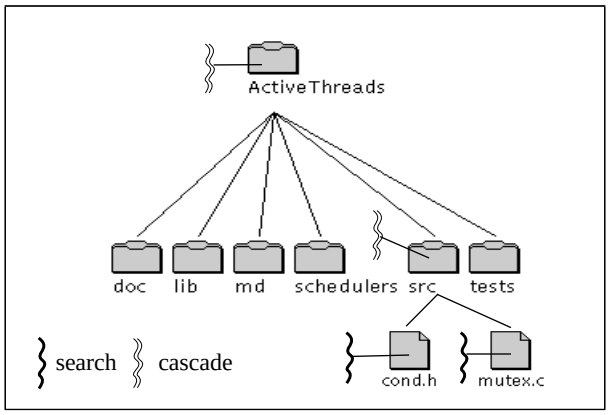

Figure 6: Thread structure of at_grep.

The figure displays a snapshot of at_grep execution with the root of the Active Threads source tree passed as an argument. It shows the mapping between the two kinds of threads

5. The original sources are available from Sun Microsystems at http://www.sun.com/workshop/sig/threads/apps.html

6. The original version used Solaris threads which have a large thread creation overhead and a complex work-bag structure was used to avoid thread creation. With the much lower thread creation overhead of Active Threads, this complexity was no longer necessary and was eliminated

and files and directories.

Since the thread creation and management overhead in Active Threads is low, no attempt was made to influence the scheduling between the search and cascade threads in the SMP implementation. In all our experiments, this simple design resulted in close to linear speedups on stand-alone SMPs.

Steal-based migration policy

We recompiled our application for the distributed platform. The sources remained essentially unmodified - a single line change to a thread creation primitive designated search threads as migratable and points where migration was most desirable were marked. In the absence of any user migration specifications, Active Threads employ a simple thread stealing policy - whenever a processor is idle it tries to steal work from any remote node which has available work. The termination condition is detected when nodes have no threads and there are no pending network messages.

These minimal changes resulted in a fairly competitive distributed implementation of grep. Figure 7 presents the speedup due to thread stealing for the command "at_grep mutex" invoked on the Active Threads source tree[7].

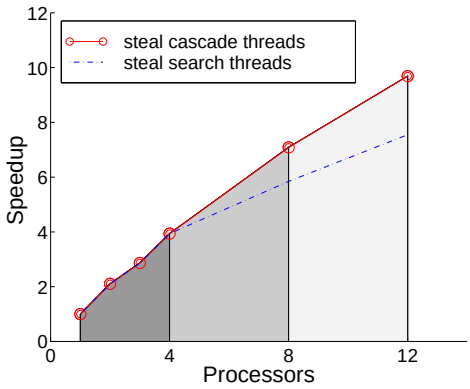

Figure 7: at_grep and thread stealing

In Figure 7, the effect of each additional multiprocessor on performance is indicated by shading the corresponding area under the speedup curve.

Since the search threads are leaves of the thread tree, they generate no additional work after migration. Cascade threads, on the other hand, generate more work and will therefore keep the recipient busy for longer. The program was therefore modified to designate cascade threads rather than the search threads as migratable. This version still relies on the default Active Threads stealing policy for migration. Although this change means potentially greater load imbalance since sharing of tasks is performed at coarser granularity, all our trial runs resulted in better performance as shown in Figure 7. Migrating cascade threads reduces the total number of migrated threads - once a thread is migrated, it generates more work for a new home node. We will shortly see that despite the fair amount of I/O, at_grep threads are short lived. Therefore, reducing the total migration overhead proved beneficial in spite of the potentially greater load imbalance.

7. Roughly 400 files in 50 directories

Push-based migration policy

To illustrate the flexibility of the Active Threads migration interface and to possibly achieve even greater performance, we next investigated taking direct control over thread migration from the runtime system by inserting explicit thread push calls.

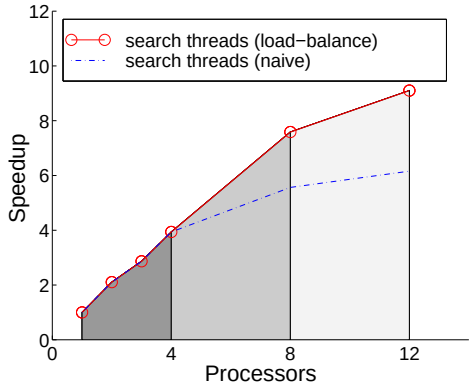

Figure 8: at_grep and thread push policies.

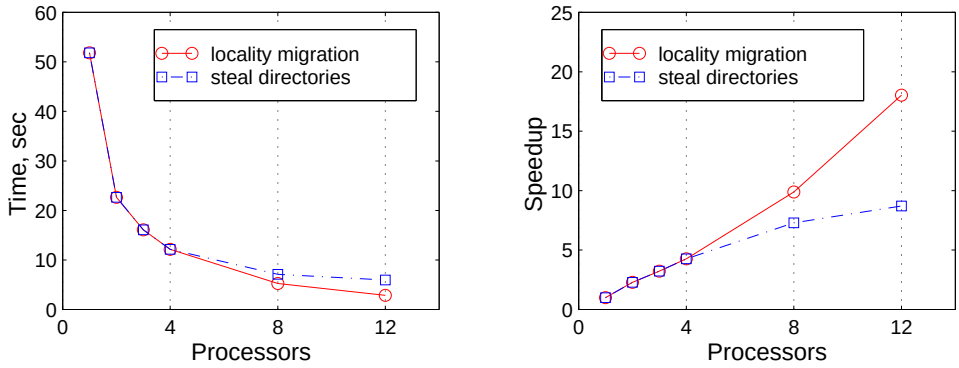

Figure 9: Effects of locality-guided thread migration.

Two heuristics were investigated:

1. naive round-robin - a single node serves as a producer, all other nodes are consumers. The producer node finds files, creates threads and pushes them to other nodes in a round-robin fashion.
2. simple load-balancing - similar to the above policy, but the producer node keeps track of the load of other nodes by recording the file sizes and destination nodes for all previously pushed search threads.

Figure 8 demonstrates the performance of at_grep under both push heuristics. While the load-balancing heuristic resulted in performance close to that of the best stealing heuristics, no further improvement was obtained. While simple, our load-balancing policy could be improved at the cost of introducing extra complexity. Instead of exploring this avenue, we have chosen to exploit the file location information to guide thread migration.

Locality-guided migration

All the multiprocessors in our network have local disks. While searching through several home directories or large file repositories that span multiple disks, grep routinely examines files located remotely. In fact, we feel that most applications of grep in a networking environment follow this pattern. Exploiting file location information for thread migration so as to perform search locally to data results in significant performance gains.

We have modified the push-based version to exploit file location information. The basic structure of the application remains unchanged. The search and cascade threads are created for files and directories as was done previously. However, if after start-up the search or cascade thread determines that data resides on a remote workstation, it pushes itself to that machine.

Figure 9 compares performance of the locality guided policy with that of the best stealing policy. Using file location information to guide thread migration resulted in significant performance improvements. We observed super-linear speedups while searching through large distributed source repositories. This is possible since, in the single processor case, most of data resides remotely while in the distributed case most data is read from the local disks.

Figure 9 also presents the absolute time in seconds for our trial runs of at_grep. A combination of thread migration with file location information allowed us to keep grep as an interactive tool even for fairly extensive searches. For instance, in our experiments, single processor execution takes almost a minute, while a distributed locality-guided execution takes about 2 seconds for the same inputs and patterns. In the case of grep, this reflects a qualitative step from batch to interactive mode.

Thread granularity and migration

Figure 10 shows the thread lifetime distribution for two cases: a) data lives on the local

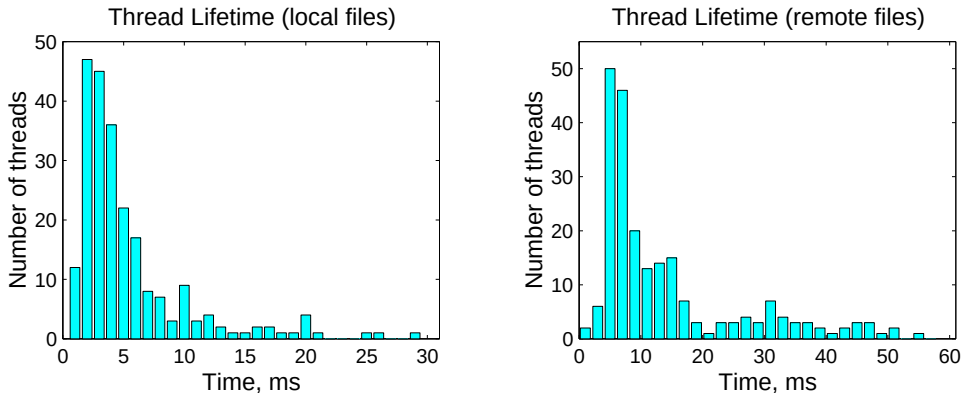

Figure 10: Thread lifetime distribution for at_grep.

disk, b) data lives remotely. The histograms cover 95% of all thread lifetimes. The threads are quite fine-grained in both the local and remote cases, though the median remote latency (9.97 ms) is more than twice as high as the median local latency (3.98 ms). This reflects the cost of going over the network to service a disk access.

It is clear that the cost of migration must be significantly lower than the average thread life-time in order to make thread migration worthwhile. In fact, with any of the other thread migration systems listed in Section 7.3, thread migration may not produce useful results for this example[8]. Since the granularity at which thread migration is useful is determined by the thread migration overhead, high-performance migration can have a qualitative effect on the programming style needed to exploit migration.

8.2 Adaptive Quadrature

The second application is adaptive quadrature, a method used for numerical integration. The value of integral

$$\int_a^b f(x)dx = I(f) \tag{1}$$

is computed within a given error tolerance ET. The domain $[a,b]$ is divided into a set of subdomains $[x_i,x_{i+1}]$ with $a = x_0 < x_1 < \ldots < x_n = b$. According to the Simpson rule, $I(f)$ is approximated by $A(f)$:

$$A(f) = \frac{1}{6}\sum_{i=1}^{n}(x_i - x_{i-1})\left[f(x_{i-1}) + 4f\left(\frac{x_{i-1}+x_i}{2}\right) + f(x_i)\right] \tag{2}$$

Now, the tolerance requirement $|I(f) - A(f)| < ET$ is met, if

$$\forall i\colon 4\ f\left(\frac{x_{i-1}+x_i}{2}\right) - f(x_{i-1}) - f(x_i) \ < \ 2ET\frac{x_i - x_{i-1}}{b-a} \tag{3}$$

holds and if the requirement specified in [MS75] holds for derivatives of f.

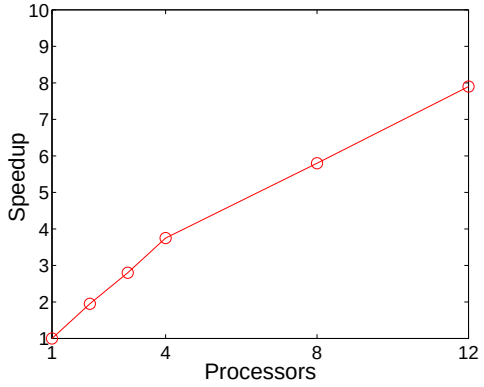

Figure 11: Adaptive Quadrature: speedup

For adaptive quadrature the domain $[a,b]$ is recursively split into subdomains until (3) holds. A split operation divides a domain into two subdomains of equal size. Our multi-threaded implementation with n threads initially divides $[a,b]$ into n subdomains of equal size. Each threads integrates f in one of these subdomains and the final result is the sum of all thread's results.

As with at_grep, the thread lifetime of this application varies significantly, because the

8. On the faster processors used in those systems, thread lifetimes would be even lower.

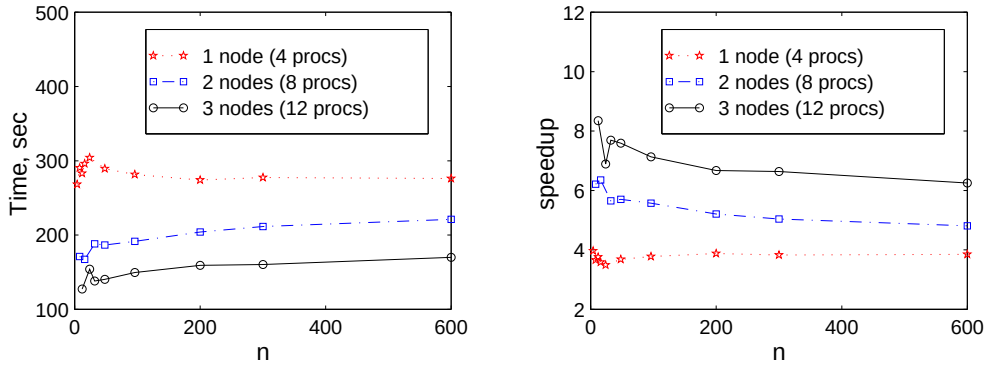

Figure 12: Runtime and speedup for integration of $f(x) = 10 \cdot \sin\left(\frac{1}{0.00001 + 1000 \cdot \sin(20x)}\right)$ by adaptive quadrature.

number of required split steps per thread depends on local properties of f. The lifetime could be predicted, but in general prediction is as expensive as the integration itself. With all threads being independent, an arbitrary initial distribution is possible. All threads are created on the same cluster and distribution is accomplished using only thread migration initiated by the stealing mechanism. Thus, the programmer's job is greatly simplified as compared to using explicit distribution. Figure 11 shows the application speedup, while Figure 12 shows the speedup with respect to the degree of parallelism of the adaptive quadrature algorithm (n). All integrations were made over the domain $(0,2]$ with $ET = 0.00001$.

9 CONCLUSIONS

We have demonstrated a thread migration system which achieves high performance by closely integrating user level threads with user level messaging. No compiler level support is necessary for pointer identification, resulting in a portable system that may be used for application programming or as a compilation target. The thread system defines an extensible event mechanism which permits a close integration between the thread and messaging systems, while maintaining the modularity of both components.

We have also defined a set of performance metrics, that make a clear distinction between the latency and the overhead of migration. Our micro-benchmarks indicate that the system is at least an order of magnitude better than existing systems. The performance would be further improved on more modern hardware with the faster processors available today.

The migration system is implemented as a set of library calls that move threads, independent of any migration policy. We use the application at_grep to explore the effect of different migration policies, particularly locality based scheduling. The applications demonstrate that high performance is obtained for both the irregular, fine-grained applications (grep and adaptive quadrature) that we investigated.

Active Threads were used as a compilation target for Sather, an explicitly parallel object-oriented language, and for a distributed extension of C++.

REFERENCES

BCF+95 Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov. N. Seizovic, and Wen-King Su. *Myrinet: A Gigabit-per-second Local Area Network.* in IEEE Micro, 15(1), February 1995.

Ber91 Brian N. Bershad.T*he Presto User Manual*. Oct 91. Available from: www.cs.washington.edu/research/compiler/papers.d/presto.html

BJK+96 Robert D. Blumofe, Cristopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Juli Zhou. *Cilk: An Effcient Multithreaded Runtime System*, in Journal of Parallel and Distributed Computing, Vol. 37. No 1, August 1996. pp 55-69.

CAL89 Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, Richard J. Littlefield. *The Amber System: Parallel Programming on a Network of Multiprocessors*, in ACM Symposium on Operating System Priciples, December 1989

CHM97 David Cronk, Matthew Haines, Piyush Mehrotra. *Thread Migration in the Presence of Pointers,* to appear in: Proceedings of the Mini_track on Multithreaded Systems, 30th Hawaii International Conference on System Science, January 1997

CKO+94 Jeremy Casa, Ravi Konuru, Steve W. Otto, Robert Prouty, Jonathan Walpole. *Adaptive Migration systems for PVM*, in Proceedings of Supercomputing '94, pp 390 - 399, Washington D.C., November 1994

FKT+96 Ian Foster, Carl Kesselman, Steven Tuecke. *The Nexus Approach to Integrating Multithreading and Communication*, in ournal of Parallel and Distributed Computing, Vol. 37. No 1, August 1996. pp 70-82.

GSD96 Seth Copen Godlstein, Klaus Erik Schauser, Dvaid E. Culler. *Lazy Threads: Implementing a Fast Parallel Call*, in Journal of Parallel and Distributed Computing, Vol. 37. No 1, August 1996. pp 5-20.

Hol97 Michael Holtkamp. *Thread Migration with Active Threads.* International Computer Science Institute Technical Report TR-97-038.

ISW97 Ayal Itzkovitz, Assaf Schuster, Lea Wolfovich. *Thread Migration and its Applications in Distributed Shared Memory Systems,* to appear in: The Journal of Systems and Software, 1997

LFA96 David K. Lowenthal, Vincent W. Freeh, Gregory R. Andrews. *Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing,* in Journal of Parallel and Distributed Computing, Vol. 37. No 1, August 1996. pp 41-54

JLHB88 Eric Jul, Henry Levy, Norman Hutchinson, Andrew Black. *Fine-Grained Mobility in the Emerald System*, in: ACM Transactions on Computer Systems, Vol. 6, No. 1, pp 109 - 133, February 1988

MR96 Edward Mascarenhas, Vermon Rego. *Ariadne: Architecure of a Portable Threads System Supporting Thread Migration*, in Software - Practice and Experience, Vol 26(3), pp 327 - 356, March 1996

MS75 M.A. Malcolm, R.B. Simpso. *Local versus Global Strategies for Adaptive Quadrature*, ACM Trans. on Mathematical Software 1(2), pp. 129-146, 1975.

NM97 R. Namyst and J. Mehaut. *PM^2: Parallel Multithreaded Machine. A computing environment for distributed architectures.* www.ffl.fr/~nemyst/pm2.html

PKBS+91 M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein and M. Weeks. *Solaris SunOS5.0 Multithread Architecture.* White paper from Sun Microsystems, Mountain View, CA.

QW97 Jürgen W. Quittek, Boris Weissman. *Efficient Extensible Synchronization in Sather.* To appear in The 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97), Decmber 1997.

SO96 D. Stoutamire and S. Omohundro. *The Sather 1.1 Specification.* International Computer Science Institute Technical Report TR-96-012.

vCGS92 T. von Eichen, D. Culler, S. Golstein and E. Schauser. *Active Messages: a mechanism for integrated communication and computation*, in Proceedings of the 19th International Conference on Computer Architecture 1992.

WCD+95 Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, Katherine Yelick. *Runtime Support for Portable Distributed Data Structures*, in Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers, May 1995.

Wei97 Boris Weissman. *Active Threads: an Extensible and Portable Light-Weight Thread System.* International Computer Science Institute Technical Report TR-97-036