Appears in the Proceedings of "SC98: High Performance Networking and Computing", November 1998.

Pthreads for Dynamic and Irregular Parallelism

Girija J. Narlikar
narlikar@cs.cmu.edu
http://www.cs.cmu.edu/~girija

Guy E. Blelloch
blelloch@cs.cmu.edu
http://www.cs.cmu.edu/~guyb

CMU School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract

High performance applications on shared memory machines have been typically written in a coarse grained style, with one heavyweight thread per processor. In comparison, programming with a large number of lightweight, parallel threads has several advantages, including simpler coding for programs with irregular and dynamic parallelism, and better adaptability to a changing number of processors. The programmer can express a new thread to execute each individual parallel task; the implementation dynamically creates and schedules these threads onto the processors, and effectively balances the load. However, unless the threads scheduler is designed carefully, the parallel program may suffer poor space and time performance.

In this paper, we study the performance of a native, lightweight POSIX threads (Pthreads) library on a shared memory machine running Solaris; to our knowledge, the Solaris library is one of the most efficient user-level implementations of the Pthreads standard available today. To evaluate this Pthreads implementation, we use a set of parallel programs that dynamically create a large number of threads. The programs include dense and sparse matrix multiplies, two N-body codes, a data classifier, a volume rendering benchmark, and a high performance FFT package. We find the existing threads scheduler to be unsuitable for executing such programs. We show how simple modifications to the Pthreads scheduler can result in significantly improved space and time performance for the programs; the modified scheduler results in as much as 44% less running time and 63% less memory requirement compared to the original Pthreads implementation. Our results indicate that, provided we use a good scheduler, the rich functionality and standard API of Pthreads can be combined with the advantages of dynamic, lightweight threads to result in high performance.

Keywords: Multithreading, Pthreads, space efficiency, dynamic scheduling, irregular parallelism, lightweight threads.

1 Introduction

Recently, shared memory multiprocessors have been used to implement a wide range of high performance applications [16, 20, 50, 55, 56]. The use of multithreading to program such applications is becoming popular, and POSIX threads or Pthreads [27] are now a standard supported on most platforms.

Pthreads may be implemented either at the kernel level, or as a user-level threads library. Kernel-level implementations require a system call to execute most thread operations, and the threads are scheduled by the kernel scheduler. This approach provides a single, uniform

thread model with access to system-wide resources, at the cost of making thread operations fairly expensive. In contrast, most operations on user-level threads, including creation, synchronization and context switching, can be implemented in user space without kernel intervention, making them significantly cheaper than kernel threads. Thus parallel programs can be written with a large number of lightweight, user-level Pthreads, leaving the scheduling and load balancing to the threads implementation, and resulting in simple, well structured, and architecture-independent code. A user-level implementation also provides the flexibility to choose a scheduler that best suits the application, independent of the kernel scheduler.

Lightweight and efficient user-level Pthreads packages are now available for shared memory programming [32, 51, 18]. However, most multithreaded parallel programs are still coded in a coarse-grained manner, with one thread per processor. Most programmers do not believe they will get high performance by simply expressing the parallelism as a large number of threads and leaving the scheduling and load balancing to the threads implementation. If threads accessing common data are scheduled on different processors, poor locality may limit the performance of the program. Further, the Pthreads scheduler may not be well suited to manage a large number of threads. Thus even multithreaded programs on shared memory machines are typically restricted to the SPMD programming style (*e.g.*, [54, 55]), and do not take advantage of lightweight threads implementations.

In this paper, we examine the applicability of lightweight, user-level Pthreads to express dynamic or irregular parallel programs, that is, programs that can be more simply expressed using a large number of lightweight threads. Since the Solaris Pthreads library is one of the most efficient user-level implementations of Pthreads available today, we decided to study the performance of seven multithreaded, parallel programs on a Sun Enterprise 5000 SMP running Solaris 2.5. The programs include dense and sparse matrix multiplies, two N-body codes, a data classifier, a volume rendering benchmark, and a high performance FFT package. We find that the only scheduler currently available in the library implementation, which uses a FIFO queue, creates a very large number of simultaneously active threads, resulting in high memory allocation, high resource contention, and poor speedup. We then describe simple modifications we made to the Solaris Pthreads implementation to improve space and time performance. The modified version of the Pthreads implementation uses a space-efficient scheduling mechanism [35] that results in a good speedup, while keeping memory allocation low. For example, for the dense matrix multiply program, the modified Pthreads scheduler reduces the running time on 8 processors compared to the original scheduler by 44%, and the memory requirement by 63%; this allows the program to run on 8 processors within 10% of the hand-optimized, platform-specific parallel BLAS3 library. Note that lowering memory requirement often improves the parallel running time due to fewer page and TLB misses, and less time spent waiting in the kernel for system calls related to memory allocation.

The parallel programs used in our experiments were written to dynamically create a large number of Pthreads. We show that the programs, although simpler than their coarse grained versions, result in equivalent performance, provided the modified Pthreads scheduler is used. The threads in the parallel programs are expressed by the programmer with sufficient granularity to amortize thread costs and provide good locality within each thread. Our modified Pthreads implementation supports the full functionality of the original Pthreads library. Therefore, any existing Pthreads programs can be executed using our space-efficient scheduler, including programs with blocking locks and condition variables. Some previous implementations of efficient schedulers [11, 29, 35] do not support such blocking synchronizations. Our results indicate that, provided we use a good scheduler, the rich functionality

and standard API of Pthreads can be combined with the advantages of dynamic, lightweight threads to result in high performance.

The remainder of this paper is organized as follows. Section 2 summarizes the advantages of using lightweight threads and gives an overview of related work on scheduling user-level threads. Section 3 describes the native Pthreads implementation on Solaris, and presents experimental results for one of the parallel benchmarks, namely, dense matrix multiply, using the existing Pthreads library. Section 4 briefly explains each modification we made to the Pthreads implementation, along with the resulting change in space and time performance for the benchmark. Section 5 describes the remaining parallel benchmarks and compares the performance of the original version with the version rewritten to create a large number of threads, using both the original implementation and the implementation with the modified scheduler. Section 6 summarizes our results and discusses some future work.

2 Motivation and related work

Shared memory parallel programs are often written in a coarse-grained style with a small number of threads, typically one per processor. In contrast, a fine-grained program creates a large number of threads, where the number grows with the problem size, rather than the number of processors. In the extreme case, a separate thread may be created for each function call or each iteration of a parallel loop. In this paper, we focus on fine-grained Pthreads programs where threads are expressed at a more moderate granularity, to amortize thread operation costs and provide locality. However, this still allows a large number of Pthreads to be created, and has several advantages over the conventional coarse-grained style, which we summarize below.

- The programmer can create a new thread for each parallel task, without explicitly mapping the threads to processors. This results in a simpler, more natural programming style, particularly for programs with irregular and dynamic parallelism. Similarly, the implicit parallelism in functional languages, or the loop parallelism extracted by parallelizing compilers is fine grained, and can be more naturally expressed as lightweight threads.

- The resulting program is architecture independent, since the parallelism is not statically mapped to a fixed number of processors. This is particularly useful in a multiprogramming environment, where the number of processors available to the computation may vary over the course of its execution [10, 52].

- Since the number of threads expressed is much larger than the number of processors, the threads can be effectively load balanced by the implementation. The programmer does not need to implement a load balancing strategy for every application that cannot be mapped statically.

- A lightweight, user-level threads implementation can provide a number of alternate scheduling techniques, independent of the kernel scheduler. Modifying the execution order of individual parallel tasks in a fine-grained program may then simply involve switching between schedulers for the corresponding threads. In contrast, modifying this execution order in a coarse-grained program can involve considerable programming effort.

While the expression of a large number of lightweight threads has several advantages, an efficient implementation is required to schedule the threads onto the processors. The scheduler must balance the load, be space-efficient (*i.e.*,, keep memory requirements low), and ensure good locality.

In this paper, we focus on the scheduling mechanisms used in lightweight threads packages written for shared memory machines. In particular, we are interested in implementing a scheduler that efficiently supports dynamic and irregular parallelism.

2.1 Scheduling lightweight threads

A variety of lightweight, user-level threads systems have been developed [6, 11, 14, 15, 25, 29, 33, 37, 40, 45, 53], including mechanisms to provide coordination between the kernel and the user-level threads library [2, 49, 31]. Although the main goal of the threads schedulers in previous systems has been to achieve good load balancing and/or locality, a large body of work has also focused on developing scheduling techniques to conserve memory requirements. Since the programming model allows the expression of a large number of lightweight threads, the scheduler must take care not to create too many simultaneously active threads. This ensures that system resources like memory are not exhausted or do not become a bottleneck in the performance of the parallel program. For example, consider the serial execution of a simple computation, represented by the *computation graph* in Figure 1. Each node in the graph represents a computation within a thread, and each edge represents a dependency. The solid right-to-left edges represent the forking of child threads, while the dashed left-to-right edges represent joins between parent-child pairs. The vertical edges represent sequential dependencies within a single thread. Let us assume that a single global list of ready threads is maintained by the scheduler. If this list is implemented as a LIFO stack, the nodes are executed in a depth-first order. This results in as many d simultaneously active threads, where d is the maximum number of threads along any path in the graph[1]. In contrast, if the ready list were implemented as a FIFO queue, the system would execute the threads in a breadth-first order, creating a much larger number of threads. Thus a serial execution of the graph in Figure 1 using a FIFO queue would result in all 7 threads being simultaneously active, while a LIFO stack would result in at most 3 active threads.

The initial approaches to conserving memory were based on heuristics that work well for some applications, but do not provide guaranteed bounds on space [6, 19, 22, 29, 33, 39, 44, 46]. For example, Multilisp [44], a flavor of Lisp that supports parallelism through the *future* construct, uses per-processor LIFO stacks of ready threads to limit the parallelism. Lazy thread creation [22, 33] avoids allocating resources for a thread unless it is executed in parallel. Filaments [29] is a package that supports fine-grained fork-join or loop parallelism using non-preemptive, stateless threads; it further reduces overheads by coarsening and pruning excess parallelism.

Recent work has resulted in provably efficient scheduling techniques that provide upper bounds on the space required by the parallel computation [9, 11, 12, 13, 35]. Since there are several possible execution orders for lightweight threads in a computation with a high degree of parallelism, the provably space-efficient schedulers restrict the execution order for the threads to bound the space requirement. For example, the Cilk multithreaded system [11] guarantees a space bound of $p \cdot S_1$ on p processors for a program with a serial space requirement

[1]For programs with a large amount of parallelism, d is typically much smaller than the total number of threads.

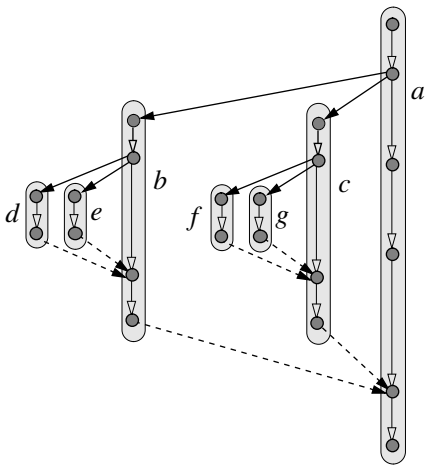


Figure 1: An example computation graph with fork-join style of parallelism. A right-to-left bold edge represents the fork of a child thread, while a left-to-right dashed edge represents a join between a parent and child; a sequential dependency within a thread is represented by a vertical edge. For example, thread e is the child of thread b, which is the child of thread a.

of S_1, by maintaining per-processor stacks of ready threads. When a processor runs out of threads on its own stack, it picks another processor at random, and steals from the bottom of its stack. Various other systems use a similar work stealing strategy [25, 33, 37, 53] to control the parallelism. In previous work [35], we presented a new, provably space-efficient scheduling algorithm that uses a shared "parallel" stack and provides a space bound of $S_1 + O(p \cdot D)$ for a program with a critical path length of D. The algorithm prioritizes threads according to their serial execution order, and preempts threads when they run out of a preallocated memory quota. We showed that this algorithm results in lower space requirements for parallel benchmarks compared to Cilk, while maintaining good performance.

In this paper, we implement a variation of the scheduling algorithm from previous work [35] in the context of Pthreads. While previous systems with space-efficient schedulers implement a restricted set of thread operations (such as fork and join), our work supports the standard and more general Pthreads API, which includes locks and condition variables. We modify an existing, native Pthreads library implementation by adding the space-efficient algorithm to its scheduler. Since the scheduling algorithm does not affect other important parts of the library, such as synchronization or signal handling, any existing Pthreads program can be run as is with our modified Pthreads scheduler. If the program is written to use a small number of threads, its performance using the new scheduler will be identical to the original scheduler. However, as we shall see later in the paper, a fine-grained program that dynamically creates and destroys a large number of Pthreads enjoys improved space and time performance with the modified implementation.

The existing Pthreads interface allows the programmer to choose between a set of alternate scheduling policies for each thread. Two of these policies, `SCHED_FIFO` and `SCHED_RR` are well defined, and must respect the priorities assigned to each thread by the user. Our space-efficient scheduler could be provided as an implementation of the third policy, `SCHED_OTHER`, which is left unspecified by the POSIX standard. This paper focuses on a prioritized implementation of the space-efficient scheduler, that is, similar to the other two policies, runnable threads assigned higher priorities by the user will be scheduled before lower priority threads.

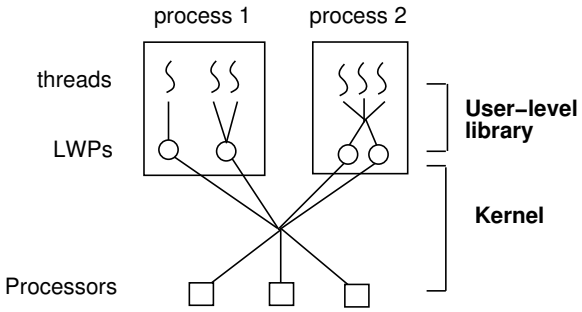


Figure 2: Scheduling of lightweight Pthreads and kernel-level LWPs in Solaris. Threads are multiplexed on top of LWPs at the user level, while LWPs are scheduled on processors by the kernel.

However, threads within a single priority level are scheduled in a space-efficient manner. Thus, our policy can be used along with the other Pthread scheduling policies in a single program, as long as all threads within any priority level use the same scheduling policy.

3 Case study: Pthreads on Solaris

In this section, we describe the native Pthreads library on Solaris 2.5, followed by some experiments measuring the performance of a parallel matrix multiply benchmark that uses Pthreads on Solaris. The experiments with the remaining benchmarks are described in Section 5.

The Solaris operating system contains kernel support for multiple threads within a single process address space [41]. One of the goals of the Solaris Pthreads implementation is to make the threads sufficiently lightweight so that thousands of them can be present within a process. The threads are therefore implemented by a user-level threads library so that common thread operations such as creation, destruction, synchronization and context switching can be performed efficiently without entering the kernel.

Lightweight, user-level Pthreads on Solaris are implemented by multiplexing them on top of kernel-supported threads called LWPs. The assignment of the lightweight threads to LWPs is controlled by the user-level Pthreads scheduler [51]. A thread may be either bound to an LWP (to schedule it on a system-wide basis) or may be multiplexed along with other unbound threads of the process on top of one or more LWPs. LWPs are scheduled by the kernel onto the available CPUs according to their scheduling class and priority, and may run in parallel on a multiprocessor. Figure 2 shows how threads and LWPs in a simple Solaris process may be scheduled. Process 1 has one thread bound to an LWP, and two other threads multiplexed on another LWP, while process 2 has three threads multiplexed on two LWPs. In this paper, we study and modify the policy used to schedule unbound Pthreads at a given priority level on top of LWPs.

Since Solaris Pthreads are created, destroyed and synchronized by a user-level library without kernel intervention, these operations are significantly cheaper compared to the corresponding operations on kernel threads. Figure 3 shows the overheads for some Pthread operations on a 167 MHz UltraSPARC processor. Operations on bound threads involve operations on LWPs and require kernel intervention; they are hence more expensive than user-level operations on unbound threads. Note, however, that the user-level thread operations are still significantly more expensive than function calls; *e.g.*, the thread creation time of $20.5\mu s$ corresponds to over 3400 cycles on the 167 MHz UltraSPARC. The Pthreads im-

Operation	Create	Context switch	Join	Semaphore sync.
Unbound thread	20.5	9	6	19
Bound thread	170	11	8.5	55

Figure 3: Uniprocessor timings in microseconds for Solaris threads operations on a 167 MHz UltraSPARC running Solaris 2.5. Creation time is with a preallocated stack, and does not include any context switch. (Creation of a bound or unbound thread without a preallocated stack incurs an additional overhead $200\mu s$ for the smallest stack size of a page(8KB). This overhead increases to $260\mu s$ for a 1MB stack.) Join is the time to join with a thread that has already exited. Semaphore synchronization time is the time for two threads to synchronize using a semaphore, and includes the time for one context switch.

plementation incurs this overhead for every thread expressed in the program, and does not attempt to automatically coarsen the parallelism by combining threads. Therefore, the overheads limit how fine-grained a task may be expressed using Pthreads without significantly affecting performance. It is left to the programmer to select the finest granularity for the threads such that the overheads remain insignificant, while maintaining portability, simplicity and load balance (see Section 5.3 for a discussion of thread granularity).

Although more expensive than function calls, the thread overheads are low enough to allow the creation of many more threads than the number of processors during the execution of a parallel program, so that the job of scheduling these threads and balancing the load across processors may be left to the threads scheduler. Thus, this implementation of Pthreads is well-suited to express medium-grained threads, resulting in simple and efficient code, particularly for programs with dynamic parallelism. For example, Figure 4 shows the pseudocode for a simple, block-based, divide-and-conquer algorithm for matrix multiplication using dynamic parallelism: each parallel, recursive call is executed by forking a new thread. To ensure that the total overhead of thread operations is not significant, the parallel recursion on a 167 MHz UltraSPARC is terminated once the matrix size is reduced to 64×64 elements; beyond that point, an efficient serial algorithm is used to perform the multiplication[2]. The total time to multiply two 1024×1024 matrices with this algorithm on a single 167 MHz UltraSPARC processor, using a LIFO scheduling queue and assuming a preallocated stack for every thread created, is $17.6s$; of this, the thread overheads are no more than $0.2s$. The more complex but asymptotically faster Strassen's matrix multiply can also be implemented in a similar divide-and-conquer fashion with a few extra lines of code; coding it with static partitioning is significantly more difficult. Further, efficient, serial, machine-specific library routines can be easily plugged in to multiply the 64×64 submatrices at the base of the recursion tree. Note that the allocation of temporary space in the algorithm in Figure 4 can be avoided, but this would significantly add to the complexity of the code or reduce the parallelism.

3.1 Performance of matrix multiply using the native Pthreads implementation

We implemented the algorithm in Figure 4 on an 8-processor Sun Enterprise 5000 SMP running Solaris with 2 GB of main memory. Each processor is a 167 MHz UltraSPARC with a 512KB L2 cache. We selected the only scheduling mechanism available in the Solaris 2.5

[2]The matrix multiply code was adapted from an example Cilk program available with the Cilk distribution [11].

```
Matrix_Mult(A, B, C, size) {
   if (size <= K) serial_mult(A, B, C, size);
   else
      T = mem_alloc(size * size);
      initialize smaller matrices as quadrants of A, B, C, and T;
      hsize = size/2;
      fork Matrix_Mult(A11, B11, C11, hsize);
      fork Matrix_Mult(A11, B12, C12, hsize);
      fork Matrix_Mult(A21, B12, C22, hsize);
      fork Matrix_Mult(A21, B11, C21, hsize);
      fork Matrix_Mult(A12, B21, T11, hsize);
      fork Matrix_Mult(A12, B22, T12, hsize);
      fork Matrix_Mult(A22, B22, T22, hsize);
      fork Matrix_Mult(A22, B21, T21, hsize);
      join with all forked child threads;
      Matrix_Add(T, C);
      mem_free(T);
}
```

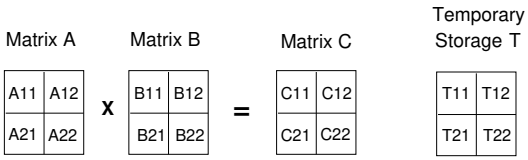


Figure 4: Pseudocode for a divide-and-conquer parallel matrix multiply (when the matrix size is a power of 2). The `Matrix_Add` function is implemented similarly using a parallel divide-and-conquer algorithm. The constant `K` to check for the base condition of the recursion is set to 64 on a 167 MHz UltraSPARC.

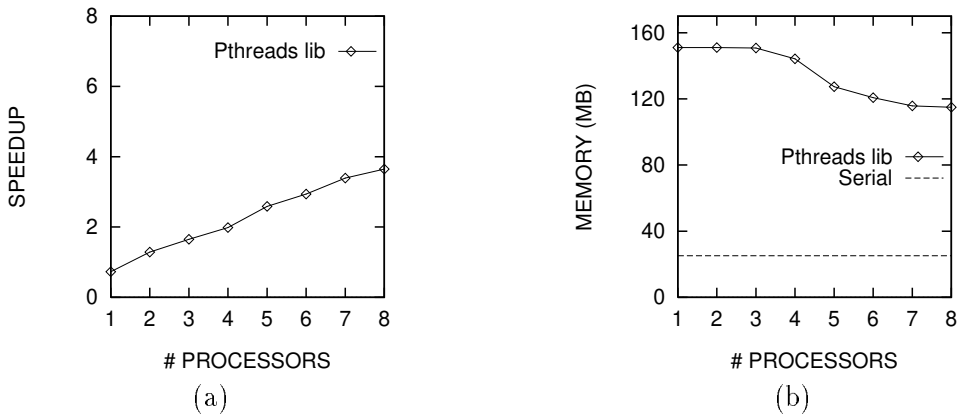


Figure 5: Performance of matrix multiply on an 8-processor Enterprise 5000 SMP using the native Pthreads implementation: (a) speedup with respect to a serial C version; (b) high water mark of total heap memory allocation during the execution of the program. "Serial" is the space requirement of the serial program, and equals the size of the input matrices.

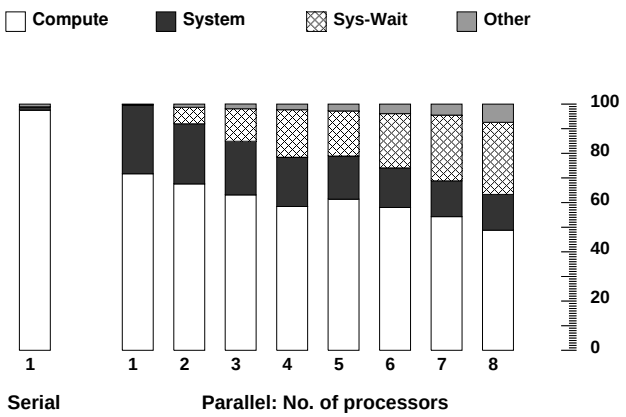


Figure 6: A breakdown of execution times on up to 8 processors for matrix multiply. "Compute" is the percentage of time doing useful computation, "system" is the percentage of time spent in system calls, and "sys-wait" is the percentage of time spent waiting in the system. "Other" includes idle time, the time spent waiting on user-level locks, and the time spent faulting in text and data pages.

Pthreads implementation, `SCHED_OTHER`, which uses a FIFO scheduling queue. Figure 5 (a) shows the speedup of the program with respect to the serial C version written with function calls instead of forks. The speedup was unexpectedly poor for a compute-intensive parallel program like matrix multiply. Further, as shown in Figure 5 (b), the maximum memory allocated by the program during its execution (*e.g.*, 115 MB on 8 processors) significantly exceeded the memory allocated by the corresponding serial program (25 MB).

To detect the cause for the poor performance of the program, we used a profiling tool (Sun Workshop version 4.0) to obtain a breakdown of the execution time, as shown in Figure 6. The processors spend a significant portion of the time in the kernel making system calls. The most time-consuming system calls were those involved in memory allocation. We also measured the maximum number of threads active during the execution of the program: the Pthreads scheduler creates more than 4500 active threads during execution on a single processor. In contrast, a simple, serial, depth-first execution of the program (in which a child preempts its parent as soon as it is forked) on a single processor should result in just 10 threads being simultaneously active. Both these measures indicate that the native Pthreads scheduler creates a large number of active threads, which all contend for allocation of stack and heap space, as well as for scheduler locks, resulting in poor speedup and high memory allocation. Note that even if a parallel program exhibits good speedups for a given problem size, it is important to minimize its memory requirement; otherwise, as the problem size increases, the performance soon begins to suffer due to excessive TLB and page misses.

The Solaris Pthreads implementation creates a very large number of active threads because it uses a FIFO queue. Further, when a parent thread forks a child thread, the child thread is added to the queue rather than being scheduled immediately. As a result, the computation graph is executed in a breadth-first manner. (This matrix multiply program has a computation graph similar to the one in Figure 1; at each stage 8 threads are forked instead of the 2 shown in the figure.)

To improve the time and space performance of multithreaded applications a scheduling technique that creates fewer active threads, as well as limits their memory allocation, is

necessary. We describe our experiments in using such a scheduling technique with the Solaris Pthreads library in the rest of the paper.

4 Improvements to the native Pthreads implementation

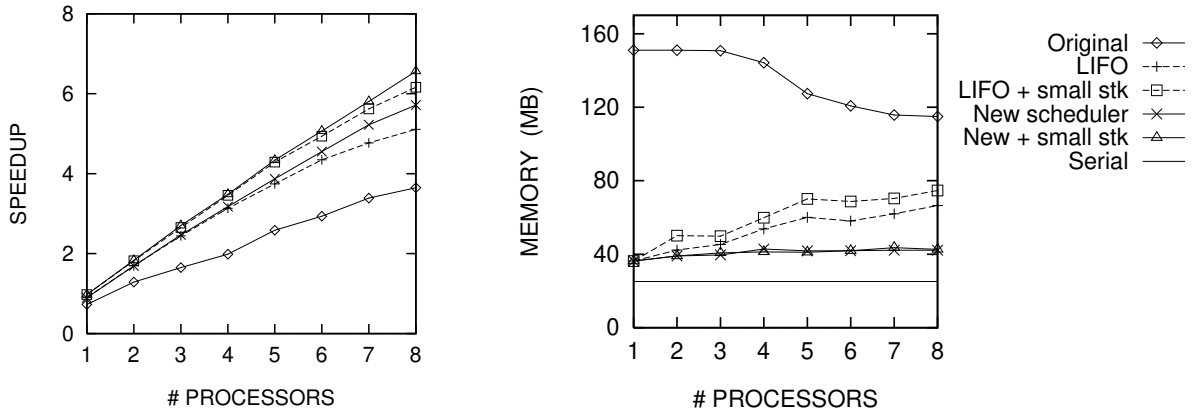
In this section, we list the modifications we made to the user-level Pthreads implementation on Solaris 2.5 to improve the performance of the matrix multiply algorithm described in Section 3. The effect of each modification on the program's space and time performance is shown in Figure 7. All the speedups in Figure 7(a) are with respect to the serial C version of matrix multiply. The curves marked "Original" in the figure are for the original Pthreads scheduler (with the deadlock-avoidance feature of automatic creation of new LWPs [51] disabled to get accurate timings). All threads were created at the same priority level, and the term "scheduling queue" used below refers to the set of all threads at that priority level.

1. **LIFO scheduler.** We modified the scheduling queue to be last-in-first-out (LIFO) instead of FIFO. A FIFO queue executes the threads in a breadth-first order, while a LIFO queue results in execution that is closer to a depth-first order. As expected, this reduces the memory requirement, though it still increases with the number of processors; the speedup also improves significantly (see curve labeled as "LIFO" in Figure 7).

2. **Space-efficient scheduler.** Next, we implemented a variation of the space-efficient scheduling technique described in [35], which provides an upper bound on the space required by a parallel computation. It guarantees that a parallel computation with a serial space requirement of S_1 and critical path length D, requires $S_1 + O(pD)$ space on p processors. The main difference between this technique and the LIFO scheduler is that threads in the scheduling queue at each priority level are always maintained in their serial, depth-first execution order. Maintaining this order at runtime is simple and inexpensive. The scheduling algorithm can be described as follows.

   - There is an entry in the scheduling queue for every thread that has been created but that has not yet exited. Thus threads represented in the queue may be either ready, blocked, or executing. These entries serve as placeholders for blocked or executing threads.
   - When a parent thread forks a child thread, the parent is preempted immediately and the processor starts executing the child thread.
   - A newly forked thread is placed to the immediate left of its parent in the scheduling queue.
   - Every time a thread is scheduled, it is allocated a memory quota (a simple integer counter) initialized to a constant K bytes. When it allocates m bytes of memory, the counter is decremented by m, and when the counter reaches zero, the thread is preempted. If a thread contains an instruction that allocates $m > K$ bytes, δ *dummy* threads (threads that perform a no-op and exit) are inserted in parallel[3] by the library before the allocation, where δ is proportional to m/K. The constant K can be used as a parameter to adjust the trade-off between space and time (see [35]).

[3]Since the Pthreads interface allows only a binary fork, these δ threads are forked as a binary tree instead of a δ-way fork.



(a) (b)

Figure 7: Performance of matrix multiply on an 8-processor Enterprise 5000 SMP using variations of the native Pthreads implementation: (a) speedup with respect to a serial C version; (b) high water mark of heap memory allocation during the execution of the program. The results were averaged over 3 consecutive runs of the program. "Original" is with the original FIFO scheduler, "LIFO" uses a LIFO scheduler, "LIFO + small stk" stands for the LIFO scheduler with a reduced default stack size, "New scheduler" uses our new space-efficient scheduler, and "New + small stk" uses the new scheduler with a reduced default stack size.

- When a thread is preempted, it is returned to the scheduling queue in the *same* position (relative to the other threads) that it was in when it was last selected for execution. This position was marked by the thread's entry in the queue.

We modified the `malloc` and `free` library functions to keep track of a thread's memory quota, and fork dummy threads if necessary. The curve labelled "New scheduler" in Figure 7(a) shows that the speedup improves with this scheduler. Further, the memory requirement (see Figure 7(b)) is significantly lower, and no longer increases rapidly with the number of processors.

3. **Reduced default stack size.** The Pthread library on Solaris specifies a stack size of 1MB for threads created with default attributes, and caches stacks of this default size for reuse. However, for applications that dynamically create and destroy a large number of threads, where each thread requires a more modest stack size, the default size of 1MB is too large. Therefore, to avoid requiring the programmer to supply and cache thread stacks in each application, we changed the default stack size to be a page (8KB); this reduces the time spent allocating the stacks. The improved performance curves are marked as "LIFO + small stk" with the LIFO scheduler, and "New + small stk" with the new, space-efficient scheduler.

   The improvements indicate that allowing the user to determine the default thread stack size may be useful. However, predicting the required stack size can be difficult for some applications. In such cases, instead of conservatively allocating an extremely large stack, a technique such as stacklets [22] or whole program optimization [24] could be used to dynamically and efficiently extend stacks.

Benchmark	Problem Size	Coarse gr	Fine gr+orig sched		Fine gr+new sched	
		Speedup	Speedup	Threads	Speedup	Threads
Matrix Mult.	1024×1024	—	3.65	1977	6.56	59
Barnes Hut	$N = 100K$, Plummer	7.53	5.76	860	7.80	34
FMM	$N = 10K$, 5 terms	—	4.90	4348	7.45	24
Decision Tree	133,999 instances	—	5.23	94	5.25	70
FFTW	$N = 2^{22}$	6.27	5.84	224	5.94	14
Sparse Matrix	30K nodes, 151K edges	6.14	4.41	55	5.96	32
Vol. Rend.	256^3 vol., 375^2 img.	6.79	5.73	131	6.72	25

Figure 8: Speedups on 8 processors over the corresponding serial C programs for the 7 parallel benchmarks. Three versions of each benchmark are listed here: the original coarse-grained version (none for Matrix Multiply, FMM or Decision Tree), the fine-grained version that uses a large number of threads with the original Solaris Pthreads scheduler, and the finer-grained version with the new, space-efficient scheduler (and an 8KB default stack size). "Threads" is the maximum number of simultaneously active threads during the 8-processor execution.

5 Experiments with other parallel benchmarks

In this section, we briefly describe our experiments with 6 additional parallel programs. The majority of them were originally written to use one thread per processor. We rewrote the programs to use a large number of Pthreads, and compared the performance of the original, coarse-grained program with the rewritten, fine-grained version using both the original Pthreads scheduler and the the new, space-efficient scheduler (using a reduced 8KB default stack size). Since Pthreads are significantly more expensive than function calls, we coarsened some of the natural parallelism available in the program. This simply involved setting the chunking size for parallel loops or the termination condition for parallel recursive calls. The coarsening amortizes thread operation costs and also provides good locality within a thread, but still allows a large number of threads to be expressed. All threads were created requesting the smallest stack size (8KB). The experiments were run on the 8-processor Enterprise 5000 described in Section 3. All programs were compiled using Sun's Workshop compiler (`cc`) 4.2, with the optimization flags `-fast -xarch=v8plusa -xchip=ultra -xtarget=native -xO4`.

5.1 The parallel benchmarks

We describe each benchmark with its experimental results; Figure 8 summarizes the results for all the benchmarks.

5.1.1 Barnes Hut

This program simulates the interactions in a system of N bodies over several timesteps using the Barnes-Hut algorithm[5]. Each timestep has three phases: an octree is first built from the set of bodies, the force on each body is then calculated by traversing this octree, and finally, the position and velocity of each body is updated. We used the "Barnes" application code from the SPLASH-2 benchmark suite [54] in our experiments.

In the SPLASH-2 Barnes code, one Pthread is created for each processor at the beginning of the execution; the threads (processors) synchronize using a barrier after each phase within

a timestep. Once the tree is constructed, the bodies are partitioned among the processors. Each processor traverses the octree to calculate the forces on the bodies in its partition, and then updates the positions and velocities of those bodies. It also uses its partition of bodies to construct the octree in the next timestep. Since the distribution of bodies in space may be highly non-uniform, the work involved for the bodies may vary to a large extent, and a uniform partition of bodies across processors leads to load imbalance. The Barnes code therefore uses a *costzones* partitioning scheme to partition the bodies among processors [48]. This scheme tries to assign to each processor a set of bodies that involve roughly the same amount of work, and are located close to each other in the tree to get better locality.

We modified the Barnes code so that, instead of partitioning the work across the processors, a new Pthread is created to execute each small, constant-sized unit of work. For example, in the force calculation phase, starting from the root of the octree, we recursively forked a new Pthread to compute forces on particles in each subtree. The recursion was terminated when the subtree had (on average) under 8 leaves. Since each leaf holds multiple bodies, this granularity is sufficient to amortize the cost of thread overheads and to provide good locality within a thread. Thus, we do not need any partitioning scheme in our code, since the large number of threads in each phase are automatically load balanced by the Pthreads scheduler. Further, no per-processor data structures were required in our code, and the final version was significantly simpler than the original code.

The simulation was run on a system of 100,000 bodies generated under the Plummer model [1] for four timesteps (as with the default Splash-2 settings, the first two timesteps were not timed). Figure 8 shows that our simpler approach achieves the same high performance as the original code. However, the library's scheduler needs to be carefully implemented to achieve this performance. Note that when the thread granularity is coarsened and therefore the number of threads is reduced, the performance of the original FIFO scheduler also improves significantly. However, as the problem size scales, unless the number of threads increases, the scheduler cannot balance the load effectively. Note that besides forks and joins, this application uses Pthread mutexes in the tree building phase, to synchronize modifications to the partially built octree.

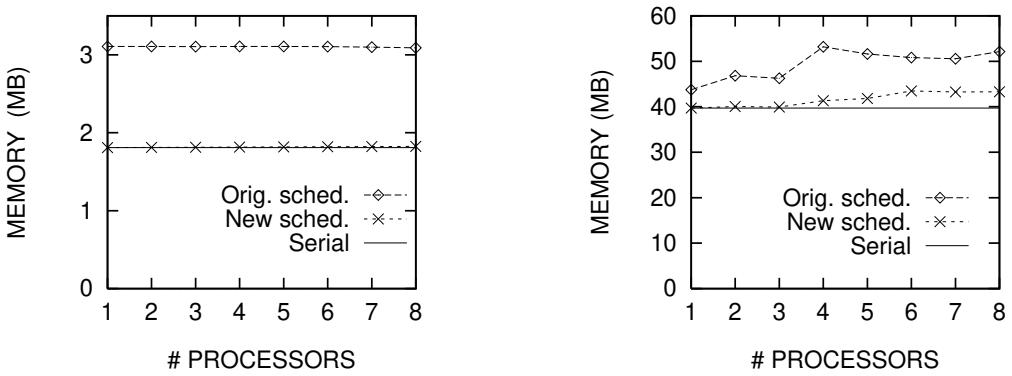
5.1.2 Fast Multipole Method

This application executes another N-Body algorithm called the Fast Multipole Method or FMM [23]. The FMM in three dimensions, although more complex, has been shown to perform fewer computations than the Barnes-Hut algorithm for simulations requiring high accuracy, such as electrostatic systems [8]. The main work in FMM involves the computation of local and multipole expansion series that describe the potential field within and outside a cell, respectively. We first wrote the serial C version for the uniform FMM algorithm, and then parallelized it using Pthreads. The parallel version is written to use a large number of threads, and we do not compare it here to any preexisting version written with one thread per processor. The program was executed on 10,000 uniformly distributed particles by constructing a tree with 4 levels and using 5 terms in the multipole and local expansions.

We describe each phase of the force calculation and how it is parallelized:

1. Multipole expansions for leaf cells are calculated from the positions and masses of their bodies; a separate thread is created for each leaf cell.

2. Multipole expansions of interior cells are calculated from their children in a bottom-up phase; a separate thread is created for each interior (parent) cell.



(a) FMM (b) Decision Tree

Figure 9: Memory requirement for the FMM and Decision Tree benchmarks. "Orig. sched." uses the native Pthreads scheduler, while "New sched." uses the new scheduler. "Serial" is the space requirement of the serial program.

3. In a top-down phase, the local expansion for each cell is calculated from its parent cell and from its well-separated neighbors; since each cell can have a large number of neighbors (up to 875), we created a separate thread to compute interactions with up to constant number (25) of a cell's neighbors. Threads are forked as a binary tree.

4. The forces on bodies are calculated from the local expansions of their leafs and from direct interactions with neighboring bodies; a separate thread is created for each leaf cell.

Since this algorithm involves dynamic memory allocation (in phase 3), we measured its space requirement with both the original and new Pthreads schedulers (see Figure 9(a)). As with matrix multiply, the new scheduling technique results in lower space requirement. The speedups with respect to the serial C version are included in Figure 8.

5.1.3 Decision Tree Builder

Classification is an important data mining problem. We implemented a decision tree builder to classify instances with continuous attributes. The algorithm used is similar to ID3 [42], with C4.5-like additions to handle continuous attributes [43]. The algorithm builds the decision tree in a top-down, divide-and-conquer fashion, by choosing a split along the continuous-valued attributes based on the best gain ratio at each stage. The instances are sorted by each attribute to calculate the optimal split. The resulting divide-and-conquer computation graph is highly irregular and data dependent, where each stage of the recursion itself involves a parallel divide-and-conquer quicksort to split the instances. We used a speech recognition dataset [26] with 133,999 instances, each with 4 continuous attributes and a true/false classification as the input. A thread is forked for each recursive call in the tree builder, as well as for each recursive call in quicksort. In both cases, we switch to serial recursion once the number of instances is reduced to 2000. Since a coarse-grained implementation of this algorithm would be highly complex, requiring explicit load balancing, we did not implement it. The 8-processor speedups obtained with the original and new schedulers are shown in

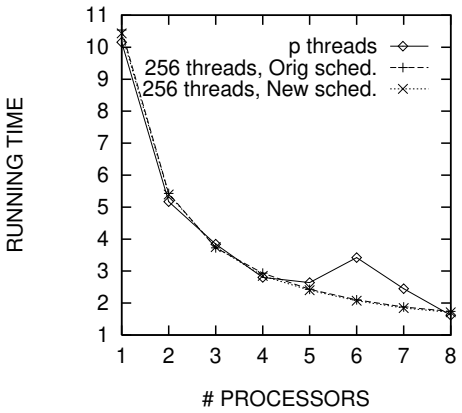


Figure 10: Running times for three versions of the multithreaded, one-dimensional DFT from the FFTW library on p processors: (1) using p threads, (2) using 256 threads with the original Pthreads scheduler, (3) using 256 threads with the modified Pthreads scheduler.

Figure 8. Both the schedulers result in good time performance; however, the new scheduler resulted in a lower space requirement (see Figure 9(b)).

5.1.4 Fast Fourier Transform

The FFTW ("Fastest Fourier Transform in the West") library [21] computes the one- and multidimensional complex discrete Fourier transform (DFT). The FFTW library is typically faster than all other publicly available DFT code, and is competitive or better than proprietary, highly optimized versions such as Sun's Performance Library code. FFTW implements the divide-and-conquer Cooley-Tukey algorithm [17]. The algorithm factors the size N of the transform into $N = N_1 \cdot N_2$, and recursively computes N_1 transforms of size N_2, followed by N_2 transforms of size N_1. The package includes a version of the code written with Pthreads, which we used in our experiments. The FFTW interface allows the programmer to specify the number of threads to be used in the DFT. The code forks a Pthread for each recursive transform, until the specified number of threads are created; after that it executes the recursion serially. The authors of the FFTW library recommend using one Pthread per processor for optimal performance.

We ran a one-dimensional DFT of size $N = 2^{22}$ in our experiments, using either p threads (where p = no. of processors), or 256 threads. Figure 10 shows the speedups over the serial version of the code for one to eight processors. Note that when p is a power of two, the problem size (which is also a power of two) can be uniformly partitioned among the processors using p threads, and being a regular computation, it does not suffer from load imbalance. Therefore, for $p = 2, 4, 8$, the version with p threads runs marginally faster. However, for all other p, the version with a larger number of threads can be better load balanced by the Pthreads implementation, and therefore performs better. This example shows that without any changes to the code, the performance becomes less sensitive to the number of processors when a large number of lightweight threads are used. The performance of this application was comparable for both the original and the modified Pthreads schedulers (see Figure 8).

5.1.5 Sparse Matrix Vector Product

We timed 20 iterations of the product $w = M \cdot v$, where M is a sparse, unsymmetric matrix and v and w are dense vectors. The code is a modified version of the Spark98 kernels [38] which are written for symmetric matrices. The sparse matrix in our experiments is generated from a finite element mesh used to simulate the motion of the ground after an earthquake in the San Fernando valley [3, 4]; it has 30,169 rows and 151,239 non-zeroes. In the coarse-grained version, one thread is created for each processor at the beginning of the simulation, and the threads execute a barrier at the end of each iteration. Each processor (thread) is assigned a disjoint and contiguous set of rows of M, such that each row has roughly equal number of nonzeroes. Keeping the sets of rows disjoint allows the results to be written to the w vector without locking.

In the fine-grained version, 128 threads are created and destroyed in each iteration. The rows are partitioned equally rather than by number of nonzeroes, and the load is automatically balanced by the threads scheduler (see Figure 8).

5.1.6 Volume Rendering

This application from the Splash-2 benchmark suite uses a ray casting algorithm to render a 3D volume [47, 54] . The volume is represented by a cube of volume elements, and an octree data structure is used to traverse the volume quickly. The program renders a sequence of frames from changing viewpoints. For each frame, a ray is cast from the viewing position through each pixel; rays are not reflected, but may be terminated early. Parallelism is exploited across these pixels in the image plane. Our experiments do not include times for the preprocessing stage which reads in the image data and builds the octree.

In the Splash-2 code, the image plane is partitioned into equal sized rectangular blocks, one for each processor. However, due to the nonuniformity of the volume data, an equal partitioning may not be load balanced. Therefore, every block is further split into tiles, which are 4×4 pixels in size. A task queue is explicitly maintained for each processor, and is initialized to contain all the tiles in the processor's block. When a processor runs out of work, it steals a tile from another processor's task queue. The program was run on a $256 \times 256 \times 256$ volume data set of a Computed Tomography head and the resulting image plane was 375×375 pixels.

In our fine-grained version, we created a separate Pthread to handle each set of 64 tiles (out of a total of 8836 tiles). Since rays cast through consecutive pixels are likely to access much of the same volume data, grouping a small set of tiles together is likely to provided better locality. However, since the number of threads created is much larger than the number of processors, the computation is load balanced across the processors by the Pthreads scheduler, and does not require the explicit task queues used in the original version. Figure 8 shows that the simpler, rewritten code runs as fast as the original code when the new scheduler is used.

5.2 Scalability

To test the scalability of our scheduling approach, we ran our benchmarks on up to 16 processors of a Sun Enterprise 6000 server. The preliminary results are similar to those in Figure 8, and can be found elsewhere [36].

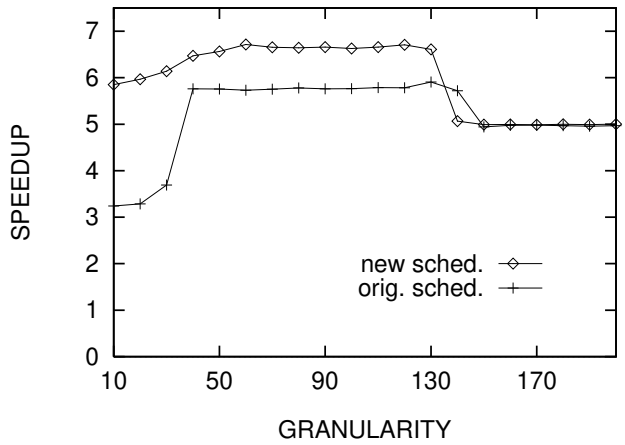


Figure 11: Variation of speedup with thread granularity (defined as the maximum number of 4×4 pixel tiles processed by each thread) for the volume rendering benchmark. "Orig. sched." is the speedup using the original FIFO scheduling queue, while "New sched." is the speedup using our space-efficient scheduler.

5.3 Selecting the optimal thread granularity

Lightweight threads allow for simple programming and good load balancing, but may incur high thread overheads and poor locality when the threads are too fine grained. Some systems support very fine-grained threads by automatically coarsening threads and executing them serially when there is sufficient parallelism [22, 29, 11]. In these systems, the thread overheads can be almost as low as a function call. However, the general Pthreads API makes implementing such optimizations difficult, and the cost of creation and destruction must be incurred for every Pthread expressed in the program. Therefore, in the experiments described so far, we adjusted the granularity of the Pthreads to amortize the cost of basic thread operations (such as creation, deletion, and synchronization). However, since our scheduler may schedule threads accessing common data on different processors, the granularity needed to be increased further for some applications to get good locality within each thread. For example, Figure 11 shows the variation of the speedup on 8 processors with thread granularity. Basic thread overheads at a finer granularity of 10 tiles per thread are at most 2.25% compared to the total execution time. Therefore, ideally, the execution on 8 processors at this granularity should be close to optimal. However, as shown in the figure, the granularity needs to be increased to around 60 tiles per thread to obtain optimal speedup. Since tiles close together in the image are likely to access common data, we expect that the slowdown at a finer granularity is mainly due to poor locality (and, to a smaller extent, due to contention on the scheduler lock). Note that, as expected, the original scheduler suffers from a bigger slowdown at the finer thread granularity, since it creates more simultaneously active threads. Further, both schedulers result in lower speedups due to load imbalance when the thread granularity is increased beyond approximately 130 tiles per thread.

Since basic Pthread operations cannot be avoided, the user must coarsen thread granularities to amortize their costs. However, ideally, we would not require the user to further coarsen threads for locality. Instead, the scheduling algorithm should schedule threads that are close in the computation graph on the same processor, so that good locality may be achieved. Then, for example, the curve in Figure 11 would not slope downwards as the granularity is reduced. We are currently working on such a space-efficient scheduling algorithm,

and preliminary results indicate that good space and time performance can be obtained even at the finer granularity that simply amortizes thread operation costs.

6 Summary and discussion

A space-efficient scheduler that limits the memory requirement of an application has the benefits of incurring fewer system calls for memory allocation, as well as fewer TLB and page misses. We have described the implementation of a simple, space-efficient scheduling technique in the context of a Pthreads library; this is the first space-efficient system supporting the general Pthreads functionality. The technique results in improved space and time performance for programs written with a large number of threads. Thus the simpler programming style of expressing a new thread to execute each unit of parallel work in programs with dynamic, irregular parallelism can be combined with the rich Pthreads functionality, to achieve high performance using our scheduling technique.

Our space-efficient scheduler maintains a globally ordered list of threads; accesses to this list are serialized by a lock. Therefore, we do not expect such a serialized scheduler to scale well beyond 16 processors. A parallelized implementation of the scheduler, such as the one described elsewhere [34], would be required to ensure further scalability.

The effectiveness of our scheduler has been demonstrated on one SMP; future work involves studying its applicability to a scalable, NUMA multiprocessor by combining it with locality-based scheduling techniques [6, 30, 40]. For example, to schedule threads on a hardware-coherent cluster of SMPs, our scheduling algorithm could be used to maintain one shared queue on each SMP, and threads would be moved between SMPs only when required.

We have shown that our space-efficient scheduler is well suited for programs with irregular and dynamic parallelism. However, programs with other characteristics may require different schedulers for high performance. For example, we did not study the performance of our scheduler for benchmarks that make extensive use of locks or condition variables. A different scheduler may be required to efficiently execute such benchmarks. Further, an important advantage of user-level threads over kernel threads is their flexibility to support multiple schedulers. Therefore, it may be necessary to simplify the task of adding new schedulers to a threads implementation. A solution is to separate the scheduler from the rest of the threads implementation via a well-defined interface, similar to previous work [7, 28].

Acknowledgements

We would like to thank the Berkeley NOW and Clumps projects for providing us access to their UltraSPARC-based workstations and Enterprise servers.

References

[1] S.J. Aarseth, M. Henon, and R. Wielen. Numerical methods for the study of star cluster dynamics. *Astronomy and Astrophysics*, 37(2):183–187, 1974.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, pages 95–109, October 1991.

[3] Hesheng Bao, Jacobo Bielak, Omar Ghattas, Loukas F. Kallivokas, David R. O'Hallaron, Jonathan R. Shewchuk, and Jifeng Xu. Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers. *Computer Methods in Applied Mechanics and Engineering*, 152(1–2):85–102, January 1998.

[4] Hesheng Bao, Jacobo Bielak, Omar Ghattas, David R. O'Hallaron, Loukas F. Kallivokas, Jonathan Richard Shewchuk, and Jifeng Xu. Earthquake Ground Motion Modeling on Parallel Computers. In *Supercomputing '96*, Pittsburgh, Pennsylvania, November 1996.

[5] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[6] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):113–121, August 1996.

[7] B. N. Bershad, E. Lazowska, and H. Levy. PRESTO : A system for object-oriented parallel programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.

[8] Guy Blelloch and Girija Narlikar. A practical comparison of n-body algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[9] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, California, July 17–19, 1995. ACM SIGACT/SIGARCH and EATCS.

[10] R. D. Blumofe and D. Papadopoulos. The performance of work stealing in multiprogrammed environments, November 1997. Draft submitted for publication, available from `http://www.cs.utexas.edu/users/rdb/papers.html`.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[12] F. W. Burton. Storage management in virtual tree machines. *IEEE Trans. on Computers*, 37(3):321–328, 1988.

[13] F. W. Burton and D. J. Simpson. Space efficient execution of deterministic parallel programs. Manuscript, December 1994.

[14] Martin C. Carlisle, Anne Rogers, John H. Reppy, and Laurie J. Hendren. Early experiences with Olden. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 1–20, Portland, Oregon, August 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.

[15] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 239–259, May 1993.

[16] Cheng Che Chen, Jaswinder Pal Singh, and Russ B. Altman. Parallel hierarchical molecular structure estimation. In *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, 1996.

[17] J. W. Cooley and J. W Tukey. An algorithm for the machine computation of complex fourier series. *Mathematics of Computation*, 19:297–301, Apr. 1965.

[18] Digital Equipment Corporation. Guide to decthreads, December 1997. available at `http://www.unix.digital.com/faqs/publications/pub_page/V40D_DOCS.HTM`.

[19] D. E. Culler and G. Arvind. Resource requirements of dataflow programs. In H. J. Siegel, editor, *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–151, Honolulu, Hawaii, May–June 1988. IEEE Computer Society Press.

[20] Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP)*, pages 206–216, 1997.

[21] Matteo Frigo and Steven G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.

[22] Seth C. Goldstein, Klaus E. Schauser, and David E. Culler. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 1995.

[23] L. Greengard. *The rapid evaluation of potential fields in particle systems*. The MIT Press, 1987.

[24] Dirk Grunwald and Richard Neves. Whole-program optimization for time and space efficient threads. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 50–59, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[25] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. Parallel programming based on continuation-passing thread. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications*, Capri, Italy, October 1994.

[26] A.G. Hauptmann, R.E. Jones, K. Seymore, M.A. Siegler, S.T. Slattery, and M.J. Witbrock. Experiments in information retrieval from spoken documents. In *BNTUW-98 Proc. DARPA Workshop on Broadcast News Understanding Systems*, February 1998.

[27] IEEE. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. IEEE/ANSI Std 1003.1, 1996 Edition.

[28] Suresh Jagannathan and Jim Philbin. A customizable substrate for concurrent languages. In *Proceedings of ACM Symposium on Programming Language Design and Implementation*, 1992.

[29] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Efficient support for fine-grain parallelism on shared memory machines. Technical Report TR 96-1, University of Arizona, January 1996.

[30] Evangelos P. Markatos and Thomas L. Blanc. Load balancing vs. locality management in shared-memory multiprocessor. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume I, Architecture, pages I:258–267, Boca Raton, Florida, August 1992.

[31] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, pages 110–121, Pacific Grove, CA, October 1991.

[32] T. Miyazaki, C. Sakamoto, M. Kuwayama, L. Saisho, and A. Fukuda. Parallel pthread library (PPL): user-level thread library with parallelism and portability. In *Proceedings of Eighteenth Annual International Computer Software and Applications Conference (COMPSAC 94)*, pages 301–306, November 1994.

[33] Eric Mohr, David Kranz, and Robert Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 1990.

[34] Girija J. Narlikar and Guy E. Blelloch. A framework for space and time efficient scheduling of parallelism. Technical Report CMU-CS-96-197, Computer Science Department, Carnegie Mellon University, 1996.

[35] Girija J. Narlikar and Guy E. Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, June 1997.

[36] Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic parallelism. Technical Report CMU-CS-98-114, Computer Science Department, Carnegie Mellon University, April 1998.

[37] Rishiyur S. Nikhil. Cid: A parallel, "shared-memory" C for distributed-memory machines. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 376–390, Ithaca, New York, August 8–10, 1994. Springer-Verlag.

[38] D. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University, October 1997.

[39] D. W. Palmer, J. F. Prins, S. Chatterjee, and R. E. Faith. Piecewise execution of nested data-parallel programs. *Lecture Notes in Computer Science*, 1033, 1996.

[40] James Philbin, Jan Edler, Otto J. Anshus, and Craig C. Douglas. Thread scheduling for cache locality. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–71, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[41] M. L. Powell, Steve R. Kleiman, Steve Barton, Devang Shah, Dan Stein, and Mary Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter 1991 USENIX Technical Conference and Exhibition*, pages 65–80, Dallas, TX, USA, January 1991.

[42] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[43] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[44] Jr. R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, 1985.

[45] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In ACM-SIGACT; ACM-SIGARCH, editor, *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, SC, July 1991. ACM Press.

[46] C. A. Ruggiero and J. Sargeant. Control of parallelism in the manchester dataflow machine. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag, Berlin, DE, 1987.

[47] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *Computer*, 27(7):45–55, July 1994.

[48] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, June 1995.

[49] E. G. Sirer, P. Pardyak, and B. Bershad. Strands: An efficient and extensible thread management architecture. Technical Report TR-97-09-01, University of Washington, Department of Computer Science and Engineering, September 1997.

[50] Deepak Srivastava and Stephen T. Barnard. Molecular dynamics simulation of large-scale carbon nanotubes on a shared-memory architecture. In *Proceedings of SC'97: High Performance Networking and Computing*, San Jose, California, November 1997.

[51] Dan Stein and Devang Shah. Implementing lightweight threads. In *Proceedings of the Summer 1992 USENIX Technical Conference and Exhibition*, pages 1–10, San Antonio, TX, 1992. USENIX.

[52] Andy Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *Operating Systems Review*, 23(5):159–66, 1989.

[53] M. T. Vandevoorde and E. S. Roberts. WorkCrews: an abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[54] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

[55] Bwolen Yang and David R. O'Hallaron. Parallel breadth-first BDD construction. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP)*, pages 145–156, June18–21 1997.

[56] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *CD-ROM Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996. IEEE.