# Pthreads for Dynamic Parallelism

Girija J. Narlikar
narlikar@cs.cmu.edu

Guy E. Blelloch
blelloch@cs.cmu.edu

April 1998
CMU-CS-98-114

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

Expressing a large number of lightweight, parallel threads in a shared address space significantly eases the task of writing a parallel program. Threads can be dynamically created to execute individual parallel tasks; the implementation schedules these threads onto the processors and effectively balances the load. However, unless the threads scheduler is designed carefully, such a parallel program may suffer poor space and time performance.

In this paper, we evaluate the performance of a native, lightweight POSIX threads (Pthreads) library on a shared memory machine using a set of parallel benchmarks that dynamically create a large number of threads. By studying the performance of one of the benchmarks, matrix multiply, we show how simple, yet provably good modifications to the library can result in significantly improved space and time performance. With the modified Pthreads library, each of the parallel benchmarks performs as well as its coarse-grained, hand-partitioned counterpart. These results demonstrate 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.

This research is supported by ARPA Contract No. DABT63-96-C-0071. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

**Keywords:** Multithreading, Pthreads, space efficiency, dynamic scheduling, nested parallelism, fine-grained parallelism, lightweight threads.

# 1 Introduction

Multithreading has become a popular methodology for expressing parallelism in the shared memory programming world. Threads are typically implemented either at the kernel level, or as a user-level library. In the former case, most thread operations involve making a system call, with the kernel being aware of the existence of multiple threads within a single process. This approach provides a single, uniform thread model with access to system-wide resources, at the cost of making thread operations (such as a creation or synchronization) fairly expensive. On the other hand, a user-level implementation allows most thread operations to execute entirely in user space without kernel intervention, making them significantly cheaper than kernel threads. Thus parallel programs can be written with a large number of such lightweight threads, leaving the job of scheduling and load balancing to the threads library, while allowing the user to write simpler, well structured, architecture-independent code. Lightweight threads are particularly useful for applications with irregular or dynamic parallelism. For example, if a program consists of a large number of tasks of varying lengths that can execute in parallel, a separate user-level thread can be created to execute each task. The threads implementation, rather than the user, schedules the threads and balances the load. Similarly, in a parallel divide-and-conquer application, even if the recursion tree is revealed at runtime, the user can create a separate thread for each recursive call to execute in parallel. Such uses of threads incur prohibitive runtime overheads if they are implemented as heavyweight, kernel-level threads. Another advantage of user-level threads is the flexibility of their implementation; since they are scheduled by the threads library, the scheduling mechanism for the threads can be independent from the kernel scheduler. In fact, the threads library can support multiple scheduling techniques, allowing the user to pick the technique that best suits his or her application.

Several lightweight, user-level threads packages have been developed [44, 6, 30, 28, 2, 10, 26, 31], which include implementations of the POSIX standard threads or *Pthreads* API [24]. Pthreads are now becoming a popular standard for expressing parallelism in the shared memory programming model. Despite the existence of lightweight implementations of Pthreads, most programmers still write applications with one or a small constant number of Pthreads per processor. The reason is performance—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 library. Thus even multithreaded programs are typically restricted to the SPMD programming style, and do not take advantage of lightweight threads implementations. Another reason for this choice of programming style may be the fact that Pthreads are not currently implemented as user level threads on all multiprocessor platforms, and therefore programs written assuming lightweight threads will not run efficiently on all platforms. We do not address this issue here, since we feel that eventually all or most platforms will provide user level threads implementations.

In this paper, we examine the applicability of lightweight, user-level Pthreads to express dynamic or irregular parallel programs, that is, the large class of programs that can be more simply expressed using a large number of lightweight threads. We study in detail the performance of the native user-level Pthreads library on Solaris 2.5 running on a Sun Enterprise SMP using the matrix multiply benchmark. We find that using the only supported scheduler, namely a FIFO queue, results in a very large number of simultaneously active threads, leading to high memory allocation and high resource contention. This prevents the compute intensive benchmark from scaling well. We then describe several simple modifications we make to the Pthreads library that improve space and time performance. The final version of the scheduler uses a provably efficient scheduling mechanism [32] that results in a good speedup for the matrix multiply benchmark, while keeping memory allocation low. The simple and portable code for matrix multiply runs within 10% of hand-optimized BLAS3 code for small matrices, and outperforms it for larger matrices. We also describe a set of 6 additional benchmarks, most of which were rewritten from their original coarse-grained versions to dynamically create a large number of Pthreads. We show that the rewritten code, although simpler than the original code, results in equivalent performance as the original code, when the modified Pthreads library was used. Thus we demonstrate 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 rest 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 library on Solaris, and presents the parallel matrix multiply benchmark with its initial performance. Section 4 briefly explains each modification we made to the Pthreads library, along with the resulting change in space and time performance. In Section 5 we describe a set of parallel benchmarks and compare the performance of the original version with the version rewritten to create a large number of threads, using both the original library and the library with the new scheduler. We summarize and discuss open problems and future work in Section 6.

# 2 Motivation and related work

Using a large number of lightweight threads has several advantages over the conventional coarse-grained style of creating one thread per processor. We summarize these advantages below.

- All the parallel tasks can be expressed as threads, without explicitly mapping the threads to processors. This results in a simpler, more natural programming style, particularly for programs with irregular and dynamic parallelism.
- 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 [9].
- 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 each application that cannot be mapped statically.
- 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.
- Since lightweight threads are implemented at the user level, the thread scheduler can be independent of the kernel scheduler. This allows the programmer to choose between a variety of alternate scheduling techniques that may be available in the threads library; adding a new scheduling mechanism to the user-level library is also easier.

In this paper, we focus on the scheduling mechanism 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 systems have been developed to schedule lightweight, dynamic threads [5, 10, 27, 40, 23, 29, 33, 46, 14, 13, 35]. Although the main goal 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 a single global list of ready threads is maintained in the system. 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]. On the other hand, implementing the ready list 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 [41, 16, 39, 5, 29, 20, 27]. For example, Multilisp [39], a flavor of Lisp that supports parallelism through the "future" construct, uses per-processor stacks of ready threads to limit the parallelism. In [5], stack and other thread resources are conserved by allocating them lazily, that is, only when the thread is first scheduled, rather than when it is created. Similarly, lazy thread creation [29, 20] avoids allocating resources for a thread unless it is executed in parallel. Filaments [27] 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 [11, 12, 10, 8, 32]. 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 [10] guarantees space-efficiency 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 [23, 33, 29, 46] to control the parallelism. In [32], we present a new, provably space-efficient scheduling algorithm that uses a shared "parallel" stack and 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 strategy from [32] in the context of Pthreads. While previous work on space-efficient scheduling supports a restricted set of thread operations and requires a specialized runtime system, in this paper we focus on the standard Pthreads API. Providing very fine-grained threads with overheads close to a function call, similar to [10, 27], makes it difficult to support a variety of user-level synchronization primitives (such as locks) that require the lightweight thread to suspend. Instead, in this work, we focus on providing an efficient scheduling mechanism to support the complete, standard Pthreads functionality, which includes locks and condition variables. We modify an existing native Pthreads library by adding the space-efficient technique to its scheduler. In fact, the existing Pthreads interface allows the programmer to choose between a set of alternate scheduling policies for each thread; thus our policy can be used along with existing scheduling policies. Since our scheduler 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 library. If the program is written to use a small number of threads, its performance will be identical to the original library. However, as we shall see later in the paper, a program that dynamically creates and destroys a large number of Pthreads enjoys improved space and time performance with the modified library.

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

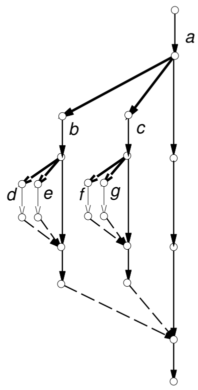



Figure 1: An example computation graph with fork-join style of parallelism. A right-to-left bold edge represents the fork of a child thread, while a left-to-right dashed edge represents a join between a parent and child; a sequential dependency within a thread is represented by a vertical edge. For example,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

# 3 Case study: Pthreads on Solaris

In this section, we describe the native Pthreads library on Solaris, followed by some experiments measuring the performance of a parallel matrix multiply benchmark that uses Pthreads on Solaris.

The Solaris 2.5 operating system contains kernel support for multiple threads within a single process address space [36][2]. The goal 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 Pthreads on Solaris are implemented by multiplexing them on top of kernel-supported threads called LWPs. The assignment of lightweight threads to LWPs is controlled by the user-level threads library [44]. 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.

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 overheads are 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 library incurs this overhead for every thread expressed in the program, and does not attempt to automatically coarsen the 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; we discuss this issue briefly in Section 6.

[2] Although Pthreads on Solaris differ from native "solaris threads" in their API, the implementations of the two threads packages on Solaris are essentially identical, so the references on solaris threads cited here apply to both the threads packages.

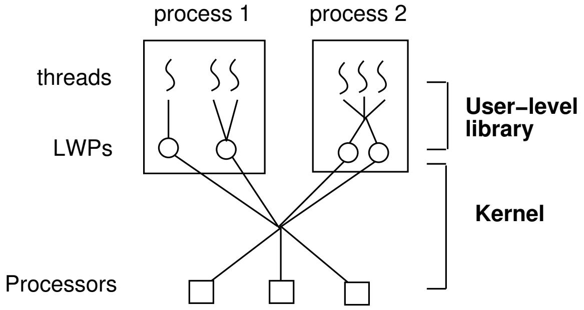



Figure 2: Scheduling of lightweight Pthreads and kernel-level LWPs in Solaris. Threads are multiplexed on top of LWPs at the user level, while LWPs are scheduled on processors by the kernel.

| 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[5], and does not include any context switch. 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.

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 library. 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 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 a size of $64 \times 64$: beyond that point, an efficient serial algorithm is used to perform the multiplication[3]. The total time to multiply two $1024 \times 1024$ 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 $17.5s$; of this, the thread overheads are no more than $0.2s$. The more complex but 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 $64 \times 64$ 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 using the native Pthreads library

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

[3]The matrix multiply code was adapted from an example Cilk program available with the Cilk distribution [10].

[5]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.

```
   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, B11, C21, hsize);
         fork Matrix_Mult(A21, B12, C22, hsize);
         fork Matrix_Mult(A12, B21, T11, hsize);
         fork Matrix_Mult(A12, B22, T12, hsize);
         fork Matrix_Mult(A22, B21, T21, hsize);
         fork Matrix_Mult(A22, B22, T22, hsize);
         join with all forked child threads;
         Matrix_Add(T, C);
         mem_free(T);
}
```

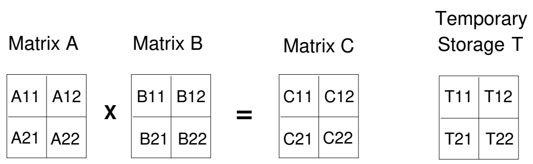
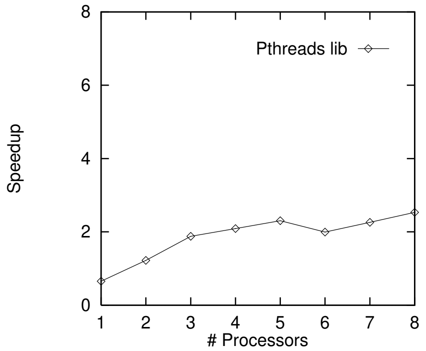
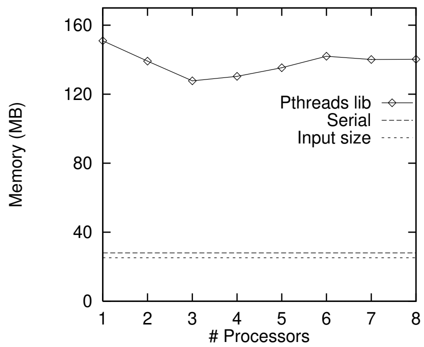



Figure 4: Pseudocode for a divide-and-conquer parallel matrix multiply. The `Matrix_Add` function is implemented similarly using a parallel divide-and-conquer algorithm. The constant `K` to check for the base condition of the recursion is set to 64 on a 167 MHz UltraSPARC.




(a)




(b)

Figure 5: Performance of matrix multiply on an 8-processor Enterprise 5000 SMP using the native Pthreads library: (a) speedup with respect to a serial C version; (b) high water mark of memory allocation during the execution of the program. "Input size" is the size of the three matrices, while "Serial" includes the additional temporary space allocated in the serial program.

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 significantly exceeded the memory allocated by the corresponding serial program[6].

To detect the cause for the poor performance of the program, we used a profiling tool[7] 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 library creates more than 3500 active threads during execution on a single processor. Note that 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 library 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 reason for the library creating a very large number of active threads is that the only scheduling technique supported by the library is 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.

[6]Note that the serial algorithm is marginally different and allocates a quarter of the temporary space at each stage, compared to the parallel version. However, this difference accounts for just a small fraction of the difference shown in Figure 5.

[7]Sun Workshop version 4.0

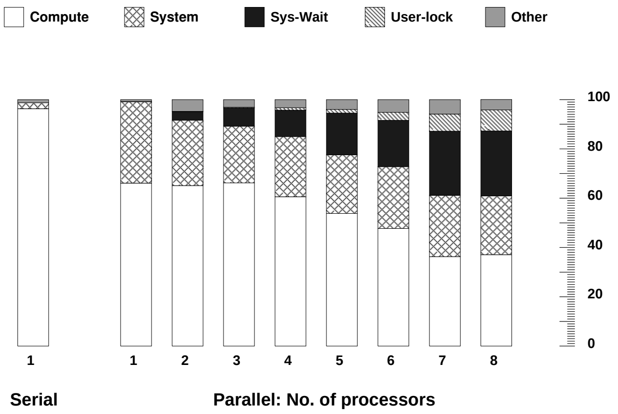



Figure 6: A breakdown of execution times on up to 8 processors for matrix multiply. "Compute" is the time doing useful computation, "system" is the time spent in system calls, "sys-wait" is the time spent waiting in the system, and "user-lock" is the time spent waiting on user-level locks.

# 4 Improvements to the native Pthreads library

In this section, we list the various modifications we made to the scheduler in the user-level Pthreads library on Solaris 2.5. The goal of these modifications was to improve the performance of the matrix multiply algorithm in Figure 4. 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. For comparison, the figure also shows the speedup obtained by the BLAS3 library routine for matrix multiplication [17] with respect to the serial C version. This library is hand-optimized by the manufacturer for the specific hardware and software system [45], and is widely considered to yield optimal performance.

The modifications on the native Pthreads library are described in the order in which they were performed:

1. **Disabling creation of new LWPs.** The current implementation of the thread library attempts to avoid deadlocks by creating new LWPs when all the existing LWPs are blocked [44]. Since in the original library, LWPs are often blocked making system calls to allocate more memory, this results in a very large number of LWPs getting created. For example, in a $1024 \times 1024$ matrix multiply on an 8-processor machine may result in as many as $40$ LWPs being created by the library. Since LWPs are kernel-level threads, they incur a larger overhead for all thread operations, including context switches. The running time on 8 processors varied from $5.8s$ to $9.97s$ depending on the total number of LWPs created, which varied from 8 to 41. Therefore, we decided to disable this deadlock-avoidance feature for the matrix multiply code by modifying the library source code. Note that this feature is not required when the LWPs are temporarily blocked in the kernel; it is, however, useful to avoid deadlocks when the LWPs are blocked in possibly indefinite, external events (such as a `poll()`). As suggested in [36], we recommend distinguishing between such blocks and allowing the user to disable the creation of new LWPs during short-term blocks. Once we fixed the number of LWPs to be the number of processors, we were able to obtain results with lower variance, and were also able to measure the speedup of the algorithm for different numbers of processors[8]. The results are shown as the curve labeled "No LWPs" in Figure 7 (this is the same curve from Figure 5 (a)).

[8]The original library would often create more than $p$ LWPs for a run on $p < 8$ processors, and the program would therefore end up using more than $p$ processors.

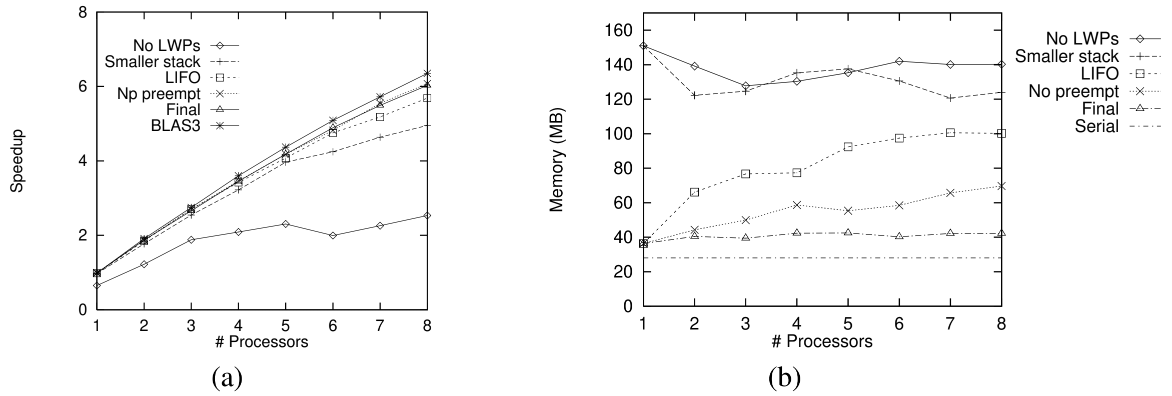



Figure 7: Performance of matrix multiply on an 8-processor Enterprise 5000 SMP using variations of the native Pthreads library: (a) speedup with respect to a serial C version; (b) high water mark of memory allocation during the execution of the program. “No LWPs” is the original library without creation of new LWPs, “Smaller stack” is with smaller 8K stacks as the default size, “LIFO” uses a LIFO scheduling queue, “No preempt” uses the space-efficient scheduler without preempting on memory allocations, and “Final” is the space-efficient scheduler with preemptions on large allocations.

2. **Reduced default stack size.** On Solaris, if a thread’s stack size is not specified while creating the thread, the default stack size is fixed at 1MB. To reduce the load on the memory allocator, the library caches a number of these stacks once threads are destroyed, so that the stacks may be reused for future threads. However, for applications that dynamically create and destroy a large number of threads, where each thread only requires a far more modest stack size, the default size of 1MB is too large. Further, a scheduling technique like FIFO creates a large number of simultaneously active threads, and therefore benefit little from caching the stacks. As a result, a large portion of time is spent in memory allocation for these stacks; this explains a portion of the large system times in Figure 6. Although programmers can supply their own stack or specify a smaller stack size, these stacks are not automatically cached by the library unless they are of the default size. We argue that to simplify the role of the programmer, the stacks should in fact be cached by the library, and therefore, there should be an option to change or specify the default stack size. We modified the default stack size to be one page or 8KB; the improved performance curves are marked as “Smaller stack” in Figure 7. Note that the memory allocation plotted in Figure 7(b) includes only the heap space, since only a very small portion of the allocated stack is actually used, while the entire allocated heap space is touched within the program. Therefore this step did not significantly improve the memory performance of the program.

   Note that it may not always be possible for the user to give a good estimate of the stack space required for the threads, which is why a large, conservative size of 1MB was chosen in Solaris. For example, the space required for a recursive function in a thread may vary depending on the input data. For such cases, an alternate strategy to conserve stack space is required to efficiently support a large number of threads, such as the one suggested in [22].

3. **LIFO scheduler.** Next, we modified the scheduling queue to be last-in-first-out (LIFO) instead of FIFO. The motivation for this change was to reduce the total number of active threads created by the library at any time during the execution. 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 reduced the memory requirements of the execution to some extent (see curve labeled as “LIFO” in Figure 7), and simultaneously improved the speedup.

4. **Space-efficient scheduler.** Finally, we implemented a variation of the space-efficient scheduling technique described in [32]. The main difference between this technique and the LIFO scheduler described above are

   - 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.
   - 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.
   - 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 counter initialized to a constant $K$ units. When it allocates $m$ bytes of memory, the counter is decremented by $m$ units, and when the counter reaches zero, the thread is preempted. If a thread contains an instruction that allocates $m > K$ bytes, $\delta$ *dummy* threads (threads that perform a no-op and exit) are inserted in parallel[9] by the library before the allocation, where $\delta$ is proportional to $m/K$. The constant $K$ can be used as a parameter to adjust the trade-off between space and time (see [32]).

   We measured the performance of matrix multiply using two versions of this technique: one without the final modification of preempting a thread due to memory allocations (labeled "No preempt" in Figure 7), and one with the preemptions (labeled "Final"). Note that both these versions yielded good speedups, and are around 10% slower than the BLAS3 routine; in fact, when the matrix size is doubled, the absolute running times of these versions are lower than BLAS3 for up to 8 processors. This is particularly encouraging since our code is high-level and platform-independent, while BLAS3 is hand-optimized. Note, however, that BLAS3 handles general matrix sizes, while our code handles sizes that are powers of 2. For a more complete implementation of a portable, divide-and-conquer matrix multiply, an algorithm such as [18] would need to be implemented.

   The two versions of the new scheduling technique differ in their memory requirements, as seen in Figure 7(b), especially as the number of processors increases. This difference is expected, since the preemptions added in the "Final" version make the scheduling technique provably space-efficient (see [32] for details). With the final scheduling technique, the performance was also less sensitive to the stack size, since fewer threads are simultaneously active, and stacks can therefore be effectively cached by the library.

# 5 Other parallel benchmarks

In this section, we describe our experiments with 6 additional parallel benchmarks. The majority of them were originally written to use one thread per processor. We rewrote the benchmarks to use a large number of Pthreads, and compared the performance of the original benchmark with the modified version using both the original Pthreads library and the library with the new scheduler. For the rewritten versions, we manually adjusted the granularity of the parallel threads (and hence the total number of threads created) to be of the finest degree such that the thread overheads did not significantly affect the running time. This adjustment is fairly simple, and involves specifying a runtime parameter that sets the chunking size for parallel loops, or the termination condition for parallel recursive calls. Since the threads are sufficiently lightweight, this

[9] Since the Pthreads interface allows only a binary fork, these $\delta$ threads are forked as a binary tree instead of a $\delta$-way fork.

| Benchmark | Original | Modified + orig. lib | | Modified + new lib | |
|---|---|---|---|---|---|
| | Speedup | Speedup | Threads | Speedup | Threads |
| Matrix Multiply | 6.35 | 2.53 | 3537 | 6.03 | 76 |
| Barnes Hut | 7.99 | 4.65 | 4185 | 7.94 | 88 |
| FMM | — | 5.18 | 2277 | 7.42 | 17 |
| Decision Tree | — | 5.03 | 92 | 5.03 | 70 |
| FFTW | 5.00 | 4.64 | 224 | 4.76 | 14 |
| Sparse Matrix | 4.74 | 4.69 | 67 | 4.98 | 9 |
| Vol. Rend. | 6.81 | 5.60 | 171 | 6.79 | 44 |

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 version (BLAS3 for Matrix Multiply; none for FMM or Decision Tree), the modified version that uses a large number of threads with the original Solaris Pthreads library, and the modified version with the Pthreads library using the new scheduling technique. "Threads" is the maximum number of active threads during the 8-processor execution.

coarsening still results in a large number of threads being created, and thus allows for automatic load-balancing. All threads were created with the smallest stack size that was sufficient to run the experiments; in most cases, a stack of one page (8KB) was used. 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 flags `-fast -xarch=v8plusa -xchip=ultra -xtarget=native -xO4`.

We describe each benchmark with its experimental results separately; Figure 8 summarizes the results. Preliminary results on 16 processors of a Sun Enterprise 6000 are presented at the end of this section.

### 5.1 Barnes Hut

This program simulates the interactions in a system of $N$ bodies over several timesteps using the Barnes-Hut algorithm[25]. 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 accordingly. We used the *Barnes* application code from the SPLASH-2 benchmark suite [47] 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 [43]. 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, a new thread is created to compute the forces on bodies in every 4 leaves in the tree[10]. Since each leaf

[10]Creating a new thread for every leaf adds a 4% overhead due to thread operations, so we coarsened the parallelism by grouping

holds multiple bodies, this granularity is sufficient to amortize the cost of thread overheads and to provide locality within a thread. Thus, each phase was parallelized by simply writing a loop to create a large number of Pthreads over the set of all leafs or all bodies[11]. Further, 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 library. In addition, 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[12]. 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 library also improves significantly. However, as the problem size scales, unless the number of threads increases, the library cannot balance the load effectively. The performance of our modified Pthreads library is not affected by the high usage of locks (Pthread mutexes) in the tree-building phase.

### 5.2 Fast Multipole Method

This application executes a different $N$-Body algorithm called the Fast Multipole Method or FMM [21]. The FMM in three dimensions, although more complex, has been shown to do less work than the Barnes-Hut algorithm for simulations requiring high accuracy, such as electrostatic systems [7]. 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 cell.

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 a constant number (50) of a cell's neighbors.

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 the original and new versions of the Pthreads library (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.

---

4 leaves together. This coarsening is not required for larger problem sizes.

[11] For the force calculation phase, since we created over 4000 threads, we rewrote the flat loop as a simple binary tree of forks to avoid serializing the creation of the threads.

[12] As with the default Splash-2 settings, the first two timesteps were not measured.

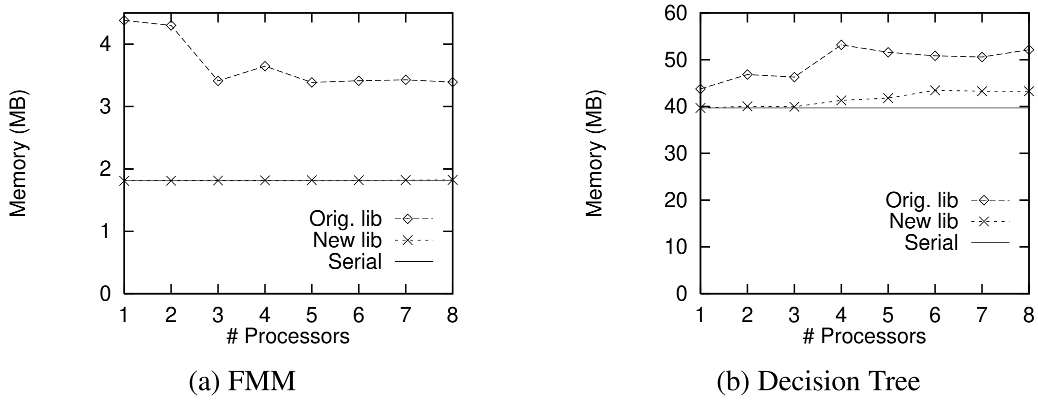



Figure 9: Memory requirement for the FMM and Decision Tree benchmarks. "Orig. lib" uses the native Pthreads library, while "New lib" uses the library modified to use the new scheduler; "Serial" is the space requirement of the serial program.

### 5.3 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 [37], with C4.5-like additions to handle continuous attributes [38]. 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 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, to minimize thread overheads, 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 library are shown in Figure 8; both the libraries result in good time performance; however, the new library resulted in a lower space requirement (see Figure 9 (b)).

### 5.4 Fast Fourier Transform

The FFTW ("Fastest Fourier Transform in the West") library [19] 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 [15]. 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 FFTW 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 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

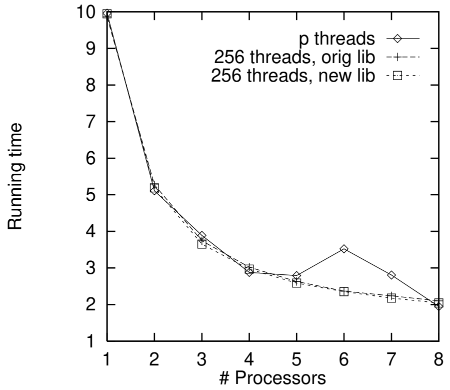



Figure 10: Running times for three versions of the multithreaded, one-dimensional DFT from the FFTW library on $p$ processors: (1) using $p$ threads, (2) using 256 threads with the original Pthreads library, (3) using 256 threads with the modified Pthreads library.

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 library, 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. For this application, the original and new versions of the Pthreads library resulted in similar performance. However, as with the volume rendering benchmark, for bigger input sizes and therefore larger numbers of threads, the difference in performance of the two libraries is expected to increase[13].

### 5.5 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 [34] 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 [4, 3]; it has 30,169 rows and 151,239 non-zeroes[14]. 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[15]. The rows are partitioned equally rather than by number of nonzeroes, and the load is automatically balanced by the threads library (see Figure 8).

[13] Our experiments were run on the largest problem size that fit into the 0.5 GB of main memory on our machine.

[14] This matrix is available as part of the Spark98 kernels.

[15] For this matrix size, creating 128 threads results in over 5% overhead compared to 8 threads; therefore we did not create any more threads.

### 5.6 Volume Rendering

This application from the Splash-2 benchmark suite uses a ray casting algorithm to render a 3D volume [47, 42] . 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 $4 \times 4$ pixels in size. They explicitly maintain a task queue for each processor, which is initialized to contain all the tiles in its own 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 $375 \times 375$ pixels.

In our version of the code, we created a separate Pthread to handle each set of 32 tiles (out of a total of 8836 tiles). Since the computation for one tile requires on average around $350\mu s$, creating one Pthread per tile would make the thread overheads significant. Further, 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. On the other hand, 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 library, and does not require the explicit task queues used in the original version. Figure 8 shows that the simpler, modified code runs as fast as the original code[16] when the modified library is used. Further, as the size of the data set grows, and therefore the number of threads created increases[17], we expect the difference in performance of the two versions of the Pthreads libraries to increase.

### 5.7 Scalability

To test the scalability of our scheduling approach, we ran our benchmarks on up to 16 processors of a Sun Enterprise 6000 server. Each processor is a faster, 250 MHz UltraSPARC with a 4MB L2 cache. The results are summarized in Figure 11. For the matrix multiply benchmark to scale beyond 12 processors, we had to rewrite the serial multiplication at the leaves of the recursion tree to avoid using `malloc`; we expect a more concurrent implementation of `malloc` will allow the original code to scale well. The timings for the Barnes-Hut benchmark do not include the tree building phase, since it does not scale well on 16 processors in the rewritten versions. We currently use the same locking tree building algorithm as the original code; therefore the use of locks increases with the number of threads. We expect a divide-and-conquer tree builder to scale better as the number of processors and threads increases. In general, the results for the parallel benchmarks were similar to those in Figure 8.

## 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

[16]The variance in running times of successive runs of the original code was much higher compared to the modified version; the times presented are the mean of 10 successive runs.

[17]Keeping the number of threads low as the problem size increases will limit its scalability and load balancing, particularly for a larger number of processors.

| Benchmark | Problem Size | Original | Modified + orig. lib | | Modified + new lib | |
|---|---|---|---|---|---|---|
| | | Speedup | Speedup | Threads | Speedup | Threads |
| Matrix Multiply | $2048^2$ matrices, $128^2$ blocks | — | 10.0 | 3537 | 14.24 | 83 |
| Barnes Hut | 200,000 bodies | 14.73 | 7.26 | 4494 | 14.95 | 92 |
| FMM | 20,000 bodies, 5 terms | — | 11.61 | 977 | 13.55 | 33 |
| Decision Tree | 133,999 instances | — | 6.02 | 101 | 6.03 | 90 |
| FFTW | $N = 2^{22}$ | 11.94 | 11.54 | 156 | 12.41 | 23 |
| Sparse Matrix | 30K nodes, 151K edges | 11.37 | 9.63 | 72 | 9.64 | 17 |
| Vol. Rend. | $256^3$ volume, $375^2$ image | 13.85 | 12.88 | 181 | 13.41 | 61 |

Figure 11: Speedups on 16 processors over the corresponding serial C programs for the 7 parallel benchmarks on a Sun Enterprise 6000 SMP. Each processor is a 250 MHz UltraSPARC with a 4MB L2 cache. Due to limited access to the machine, we could not analyze the reason behind the low speedup for the decision tree benchmark; we expect it may be due to increased bus contention or due to contention inside `malloc`. The columns are as explained in Figure 8.

implementation of a simple, space-efficient scheduling technique in the context of a POSIX threads library. 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 achieve high performance using our scheduling technique.

We have not addressed the issue of how to choose the appropriate unit of parallel work that should be executed by each thread. Since the operations in the Pthreads library that we use are significantly more expensive than a function call, we manually coarsen the parallelism for the programs described. For example, we terminate the creation of new threads in matrix multiply when the matrix size is down to $64 \times 64$; similarly, in the volume rendering benchmark, we created a Pthread to handle not one, but 32 tiles of the image plane. This coarsening ensures that thread overheads are negligible compared to the useful computation. Further, coarser threads may provide good locality within each thread, and fewer accesses to the scheduling queue may result in lower contention. For example, in the volume rendering application, if we create a separate thread to handle 8 tiles of the image (instead of 32), the serial execution of the resulting program is slowed down by $2.25\%$ due to thread overheads, while the same program on 8 processors slows down by $20\%$. We estimate that the additional slowdown is due to lower locality within each thread, as well as increased contention for the bus and the scheduling queue. Therefore, if our current scheduler were applied to a system that schedules very fine-grained threads such as [20, 27] we do not expect the performance to remain high. However, the scheduler can be extended to use globally ordered per-processor task queues. We expect this extension to scale better for fine-grained threads and larger numbers of processors by lowering contention and providing good locality within a processor, while maintaining provable space bounds.

We are currently working on practical but space-efficient ways to automatically coarsen the granularity of the unit of work scheduled across processors when per-processor queues are used. We also hope to analytically provide a trade-off between granularity and space requirement.

## Acknowledgments

We would like to thank the Berkeley NOW and Clumps projects for access to their UltraSPARC-based workstations and Enterprise servers. We also thank Sean Slattery for providing the decision tree dataset, and Joshua Simons for providing access to the Sun Enterprise 6000 server.

## 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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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`.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

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

[26] David K. Lowenthal and Gregory R. Andrews. Shared filaments: Efficient fine-grain parallelism. Technical Report TR 93-13a, University of Arizona, January 1993.

[27] 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.

[28] 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.

[29] 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.

[30] Frank Mueller. A library implementation of POSIX threads under unix. In *Proceedings of the Winter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, January 1993.

[31] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. *ACM Operating Systems Review*, 28(1):33–47, January 1994.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

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

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] Sun Microsystems, Inc. *Sun Performance Library Reference*. Part No.: 802-6439-10.

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

[47] 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.